MT XINU™

UNIX™ SUPPORT FROM BERKELEY

4.3 BSD with NFS

Programmer's
Supplementary
Documents

Volume 2

PS2

UNIX Programmer's Supplementary Documents, Volume 2 (PS2)

4.3 Berkeley Software Distribution, Virtual VAX-11 Version

April, 1986

These two volumes contain documents that supplement the manual pages in *The Unix Programmer's Reference Manual* for the Virtual VAX-11 version of the system as distributed by U.C. Berkeley.

Documents of Historical Interest

Other Languages

UNIX Programmer's Supplementary Documents
Volume 2
(PS2)

4.3 Berkeley Software Distribution
Virtual VAX-11 Version

April, 1986

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

UNIX Programmer's Supplementary Documents
Volume 2
(PS2)

4.3 Berkeley Software Distribution
Virtual VAX-11 Version

April, 1986

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

This manual reflects system enhancements made at Berkeley and sponsored in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in these documents are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

UNIX† is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

i A hierarchical file system incorporating demountable volumes,

ii Compatible file, device, and inter-process I/O,

iii The ability to initiate asynchronous processes,

iv System command language selectable on a per-user basis,

v Over 100 subsystems including a dozen languages,

vi High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

1. INTRODUCTION

There have been four versions of the UNIX time-sharing system. The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

Since PDP-11 UNIX became operational in February, 1971, over 600 installations have been put into service. Most of them are engaged in applications such as computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important characteristics of the system are

 This is a revised version of an article that appeared in Communications of the ACM, *17*, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

† UNIX is a trademark of AT&T Bell Laboratories.

its simplicity, elegance, and ease of use.

Besides the operating system proper, some major programs available under UNIX are

C compiler
Text editor based on QED[1]
Assembler, linking loader, symbolic debugger
Phototypesetting and equation setting programs[2,3]
Dozens of languages including Fortran 77, Basic, Snobol, APL, Algol 68, M6, TMG, Pascal

There is a host of maintenance, utility, recreation and novelty programs, all written locally. The UNIX user community, which numbers in the thousands, has contributed many more programs and languages. It is worth noting that the system is totally self-supporting. All UNIX software is maintained on the system; likewise, this paper and all other documents in this issue were generated and formatted by the UNIX editor and text formatting programs.

II. HARDWARE AND SOFTWARE ENVIRONMENT

The PDP-11/70 on which the Research UNIX system is installed is a 16-bit word (8-bit byte) computer with 768K bytes of core memory; the system kernel occupies 90K bytes about equally divided between code and data tables. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 96K bytes of core altogether. There are even larger installations; see the description of the PWB/UNIX systems,[2,3] for example. There are also much smaller, though somewhat restricted, versions of the system.[3]

Our own PDP-11 has two 200-Mb moving-head disks for file system storage and swapping. There are 20 variable-speed communications interfaces attached to 300- and 1200-baud data sets, and an additional 12 communication lines hard-wired to 9600-baud terminals and satellite computers. There are also several 2400- and 4800-baud synchronous communication interfaces used for machine-to-machine file transfer. Finally, there is a variety of miscellaneous devices including nine-track magnetic tape, a line printer, a voice synthesizer, a phototypesetter, a digital switching network, and a chess machine.

The preponderance of UNIX software is written in the abovementioned C language.[2] Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system was about one-third greater than that of the old. Since the new system not only became much easier to understand and to modify but also included many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we consider this increase in size quite acceptable.

III. THE FILE SYSTEM

The most important role of the system is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example, symbolic or binary (object) programs. No particular structuring is expected by the system. A file of text consists simply of a string of characters, with lines demarcated by the newline character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs that use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the **root** directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the **root**. Other system directories contain all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes, "/", and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name **/alpha/beta/gamma** causes the system to search the root for directory **alpha**, then to search **alpha** for **beta**, finally to find **gamma** in **beta**. **gamma** may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name **alpha/beta** specifies the file named **beta** in subdirectory **alpha** of the current directory. The simplest kind of name, for example, **alpha**, refers to a file that itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. The UNIX system differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name "." in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory **/dev**, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file **/dev/mt**. Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same

protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a **mount** system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of **mount** is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, **mount** replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the **mount**, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the removable volume is dismounted.

3.5 Protection

Although the access control scheme is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of ten protection bits. Nine of these specify independently read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

If the tenth bit is on, the system will temporarily change the user identification (hereafter, user ID) of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program that calls for it. The set-user-ID feature provides for privileged programs that may use files inaccessible to other users. For example, a program may keep an accounting file that should neither be read nor changed except by the program itself. If the set-user-ID bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. For example, there is a system entry invokable only by the "super-user" (below) that creates an empty directory. As indicated above, directories are expected to have entries for "." and "..". The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "." and "..".

Because anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed by "Aleph-null."[2]

The system recognizes one particular user ID (that of the "super-user") as exempt from the usual constraints on file access; thus (for example), programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and "sequential" I/O, nor is any

logical record size imposed by the system. The size of an ordinary file is determined by the number of bytes written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O, some of the basic calls are summarized below in an anonymous language that will indicate the required parameters without getting into the underlying complexities. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

```
filep = open(name, flag)
```

where **name** indicates the name of the file. An arbitrary path name may be given. The **flag** argument indicates whether the file is to be read, written, or "updated," that is, read and written simultaneously.

The returned value **filep** is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a **create** system call that creates the given file if it does not exist, or truncates it to zero length if it does exist; **create** also opens the new file for writing and, like **open**, returns a file descriptor.

The file system maintains no locks visible to the user, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file that another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor that makes a copy of the file being edited.

There are, however, sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in activities such as writing on the same file, creating files in the same directory, or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file there is a pointer, maintained inside the system, that indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used:

```
n = read(filep, buffer, count)
n = write(filep, buffer, count)
```

Up to **count** bytes are transmitted between the file specified by **filep** and the byte array specified by **buffer**. The returned value **n** is the number of bytes actually transmitted. In the **write** case, **n** is the same as **count** except under exceptional conditions, such as I/O errors or end of physical medium on special files; in a **read**, however, **n** may without error be less than **count**. If the read pointer is so near the end of the file that reading **count** characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like terminals never return more than one line of input. When a **read** call returns with **n** equal to zero, the end of the file has been reached. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a terminal by use of an escape sequence that depends on the device used.

Bytes written affect only those parts of a file implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is made to grow as needed.

To do random (direct-access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

location = lseek(filep, offset, base)

The pointer associated with **filep** is moved to a position **offset** bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on **base. offset** may be negative. For some devices (e.g., paper tape and terminals) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in **location.**

There are several additional system entries having to do with I/O and with the file system that will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

IV. IMPLEMENTATION OF THE FILE SYSTEM

As mentioned in Section 3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry found thereby (the file's *i-node*) contains the description of the file:

i the user and group-ID of its owner

ii its protection bits

iii the physical disk or tape addresses for the file contents

iv its size

v time of creation, last use, and last modification

vi the number of links to the file, that is, the number of times it appears in a directory

vii a code indicating whether the file is a directory, an ordinary file, or a special file.

The purpose of an **open** or **create** system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the **open** or **create.** Thus, during a subsequent call to read or write the file, the descriptor may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made that contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is de-allocated.

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the i-node of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\cdot 512]$ bytes. Once opened, bytes numbered below 5120 can be read with a single disk access; bytes in the range 5120 to 70,656 require two accesses; bytes in the range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last 12 device address words are immaterial, and the first specifies an internal *device name*, which is interpreted as a pair of numbers representing, respectively,

a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar terminal interfaces.

In this environment, the implementation of the **mount** system call (Section 3.4) is quite straightforward. **mount** maintains a system table whose argument is the i-number and device name of the ordinary file specified during the **mount**, and whose corresponding value is the device name of the indicated special file. This table is searched for each i-number/device pair that turns up while a path name is being scanned during an **open** or **create**; if a match is found, the i-number is replaced by the i-number of the root directory and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a **write** call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the **write** call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program that reads or writes files in units of 512 bytes has an advantage over a program that reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. If a program is used rarely or does no great volume of I/O, it may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example, verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, because it need only scan the linearly organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, because all directory entries for a file have equal status. Charging the owner of a file is unfair in general, for one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. Many installations avoid the issue by not charging any fees at all.

V. PROCESSES AND IMAGES

An *image* is a computer execution environment. It includes a memory image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory; during the execution of other processes it remains in main memory unless the appearance of an active, higher-priority process forces it to be swapped out to the disk.

The user-memory part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first hardware protection byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the stack pointer fluctuates.

5.1 Processes

Except while the system is bootstrapping itself into operation, a new process can come into existence only by use of the **fork** system call:

processid = fork( )

When **fork** is executed, the process splits into two independently executing processes. The two processes have independent copies of the original memory image, and share all open files. The new processes differ only in that one is considered the parent process: in the parent, the returned **processid** actually identifies the child process and is never 0, while in the child, the returned value is always 0.

Because the values returned by **fork** in the parent and child process are distinguishable, each process may determine whether it is the parent or child.

5.2 Pipes

Processes may communicate with related processes using the same system **read** and **write** calls that are used for file-system I/O. The call:

filep = pipe( )

returns a file descriptor **filep** and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the **fork** call. A **read** using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see Section 6.2), it is not a completely general mechanism, because the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

execute(file, arg_1, arg_2, ... , arg_n)

which requests the system to read in and execute the program named by **file**, passing it string arguments $\mathbf{arg_1}$, $\mathbf{arg_2}$, ... , $\mathbf{arg_n}$. All the code and data in the process invoking **execute** is replaced from the **file**, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because **file** could not be found or because its execute-permission bit was not set, does a return take place from the **execute** primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call:

processid = wait(status)

causes its caller to suspend execution until one of its children has completed execution. Then **wait** returns the **processid** of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly:

exit(status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. The parent is notified through the **wait** primitive, and **status** is made available to it. Processes may also terminate as a result of various illegal actions or user-generated signals (Section VII below).

VI. THE SHELL

For most users, communication with the system is carried on with the aid of a program called the shell. The shell is a command-line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. (The shell is described fully elsewhere,[3] so this section will discuss only the theory of its operation.) In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 ... arg_n

The shell splits up the command name and the arguments into separate strings. Then a file with name **command** is sought; **command** may be a path name including the "/" character to specify any file in the system. If **command** is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file **command** cannot be found, the shell generally prefixes a string such as **/bin/** to **command** and attempts again to find the file. Directory **/bin** contains commands intended to be generally used. (The sequence of directories to be searched may be changed by user request.)

6.1 Standard I/O

The discussion of I/O in Section III above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the shell, however, start off with three open files with file descriptors 0, 1, and 2. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's terminal. Thus programs that wish to write informative information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs that wish to read messages typed by the user read this file.

The shell is able to change the standard assignments of these file descriptors from the user's terminal printer and keyboard. If one of the arguments to a command is prefixed by ">", file descriptor 1 will, for the duration of the command, refer to the file named after the ">". For example:

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command:

ls >there

creates a file called **there** and places the listing there. Thus the argument **>there** means "place output on **there**." On the other hand:

ed

ordinarily enters the editor, which takes requests from the user via his keyboard. The command

ed <script

interprets **script** as a file of editor commands; thus **<script** means "take input from **script**."

Although the file name following "<" or ">" appears to be an argument to the command, in fact it is interpreted completely by the shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the

standard file descriptors 0 and 1 where appropriate.

File descriptor 2 is, like file 1, ordinarily associated with the terminal output stream. When an output-diversion request with ">" is specified, file 2 remains attached to the terminal, so that commands may produce diagnostic messages that do not silently end up in the output file.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

```
ls | pr -2 | opr
```

ls lists the names of the files in the current directory; its output is passed to **pr**, which paginates its input with dated headings. (The argument "-2" requests double-column output.) Likewise, the output from **pr** is input to **opr**; this command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by:

```
ls >temp1
pr -2 <temp1 >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the **ls** command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as **ls** to provide such a wide variety of output options.

A program such as **pr** which copies its standard input to its standard output (with processing) is called a *filter*. Some filters that we have found useful perform character transliteration, selection of lines according to a pattern, sorting of the input, and encryption and decryption.

6.3 Command separators; multitasking

Another feature provided by the shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons:

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by "&," the shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example:

```
as source >output &
```

causes **source** to be assembled, with diagnostic output going to **output**; no matter how long the assembly takes, the shell returns immediately. When the shell does not wait for the completion of a command, the identification number of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In these examples, an output file other than the terminal was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The shell also allows parentheses in the above operations. For example:

```
( date; ls ) >x &
```

writes the current date and time followed by a list of the current directory onto the file **x**. The shell also returns immediately for another request.

6.4 The shell as a command; command files

The shell is itself a command, and may be called recursively. Suppose file **tryout** contains the lines:

```
as source
mv a.out testprog
testprog
```

The **mv** command causes the file **a.out** to be renamed **testprog. a.out** is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the keyboard, **source** would be assembled, the resulting program renamed **testprog**, and **testprog** executed. When the lines are in **tryout**, the command:

```
sh <tryout
```

would cause the shell **sh** to execute the commands sequentially.

The shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It also provides general conditional and looping constructions.

6.5 Implementation of the shell

The outline of the operation of the shell can now be understood. Most of the time, the shell is waiting for the user to type a command. When the newline character ending the line is typed, the shell's **read** call returns. The shell analyzes the command line, putting the arguments in a form appropriate for **execute**. Then **fork** is called. The child process, whose code of course is still that of the shell, attempts to perform an **execute** with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the **fork**, which is the parent process, **waits** for the child process to die. When this happens, the shell knows the command is finished, so it types its prompt and reads the keyboard to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains "**&**," the shell merely refrains from waiting for the process that it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the **fork** primitive, it inherits not only the memory image of its parent but also all the files currently open in its parent, including those with file descriptors 0, 1, and 2. The shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with "<" or ">" is given, however, the offspring process, just before it performs **execute**, makes the standard I/O file descriptor (0 or 1, respectively) refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is **opened** (or **created**); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the shell need not know the actual names of the files that are its own standard input and output, because it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the shell never terminates. (The main loop includes the branch of the return from **fork** belonging to the parent process; that is, the branch that does a

wait, then reads another command line.) The one thing that causes the shell to terminate is discovering an end-of-file condition on its input file. Thus, when the shell is executed as a command with a given input file, as in:

```
sh <comfile
```

the commands in **comfile** will be executed until the end of **comfile** is reached; then the instance of the shell invoked by **sh** will terminate. Because this shell process is the child of another instance of the shell, the **wait** executed in the latter will return, and another command may then be processed.

6.6 Initialization

The instances of the shell to which users type commands are themselves children of another process. The last step in the initialization of the system is the creation of a single process and the invocation (via **execute**) of a program called **init**. The role of **init** is to create one process for each terminal channel. The various subinstances of **init** open the appropriate terminals for input and output on files 0, 1, and 2, waiting, if necessary, for carrier to be established on dial-up lines. Then a message is typed out requesting that the user log in. When the user types a name or other identification, the appropriate instance of **init** wakes up, receives the log-in line, and reads a password file. If the user's name is found, and if he is able to supply the correct password, **init** changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an **execute** of the shell. At this point, the shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of **init** (the parent of all the subinstances of itself that will later become shells) does a **wait**. If one of the child processes terminates, either because a shell found an end of file or because a user typed an incorrect name or password, this path of **init** simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence to the shell.

6.7 Other programs as shell

The shell as described above is designed to allow users full access to the facilities of the system, because it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged for.

Recall that after a user has successfully logged in by supplying a name and password, **init** ordinarily invokes the shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after log-in instead of the shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system might specify that the editor **ed** is to be used instead of the shell. Thus when users of the editing system log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on the system illustrate a much more severely restricted environment. For each of these, an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the shell. People who log in as a player of one of these games find themselves limited to the game and unable to investigate the (presumably more interesting) offerings of the UNIX system as a whole.

VII. TRAPS

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. Unless other arrangements have been made, an illegal action causes the system to terminate the process and to write its image on file **core** in the current directory. A debugger can be used to determine the state of the program at the time of

the fault.

Programs that are looping, that produce unwanted output, or about which the user has second thoughts may be halted by the use of the **interrupt** signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a **core** file. There is also a **quit** signal used to force an image file to be produced. Thus programs that loop unexpectedly may be halted and the remains inspected without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by a process. For example, the shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations that influenced the design of UNIX are visible in retrospect.

First: because we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover, such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy, but also a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, less than a page long, that buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load, with all programs, routines for dealing with each device, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no "wired-down" space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The **fork** operation, essentially as we implemented it, was present in the GENIE time-sharing system.[2] On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[2] and both the name of the shell and its general functions. The notion that the shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX.[2]

IX. STATISTICS

The following numbers are presented to suggest the scale of the Research UNIX operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

Overall, we have today:

125	user population
33	maximum simultaneous users
1,630	directories
28,300	files
301,700	512-byte secondary storage blocks used

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant *e*, and other semi-infinite problems. Not counting this background work, we average daily:

13,500	commands
9.6	CPU hours
230	connect hours
62	different users
240	log-ins

X. ACKNOWLEDGMENTS

The contributors to UNIX are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to our colleagues in the Computing Science Research Center. R. H. Canaday contributed much to the basic design of the file system. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M. D. McIlroy, and J. F. Ossanna.

References

1. L. P. Deutsch and B. W. Lampson, "An online editor," *Comm. Assoc. Comp. Mach.*, vol. 10, no. 12, pp. 793-799, 803, December 1967.
2. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.*, vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975. Reprinted as USD:26 in *UNIX User's Manual*, Usenix Association, (1986).
3. This issue, B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2115-2135, 1978.

UNIX/32V – Summary

March 9, 1979

A. What's new: highlights of the UNIX†/32V System

32-bit world. UNIX/32V handles 32-bit addresses and 32-bit data. Devices are addressable to 2^{31} bytes, files to 2^{30} bytes.

Portability. Code of the operating system and most utilities has been extensively revised to minimize its dependence on particular hardware. UNIX/32V is highly compatible with UNIX version 7.

Fortran 77. F77 compiler for the new standard language is compatible with C at the object level. A Fortran structurer, STRUCT, converts old, ugly Fortran into RATFOR, a structured dialect usable with F77.

Shell. Completely new SH program supports string variables, trap handling, structured programming, user profiles, settable search path, multilevel file name generation, etc.

Document preparation. TROFF phototypesetter utility is standard. NROFF (for terminals) is now highly compatible with TROFF. MS macro package provides canned commands for many common formatting and layout situations. TBL provides an easy to learn language for preparing complicated tabular material. REFER fills in bibliographic citations from a data base.

UNIX-to-UNIX file copy. UUCP performs spooled file transfers between any two machines.

Data processing. SED stream editor does multiple editing functions in parallel on a data stream of indefinite length. AWK report generator does free-field pattern selection and arithmetic operations.

Program development. MAKE controls re-creation of complicated software, arranging for minimal recompilation.

Debugging. ADB does postmortem and breakpoint debugging.

C language. The language now supports definable data types, generalized initialization, block structure, long integers, unions, explicit type conversions. The LINT verifier does strong type checking and detection of probable errors and portability problems even across separately compiled functions.

Lexical analyzer generator. LEX converts specification of regular expressions and semantic actions into a recognizing subroutine. Analogous to YACC.

Graphics. Simple graph-drawing utility, graphic subroutines, and generalized plotting filters adapted to various devices are now standard.

Standard input-output package. Highly efficient buffered stream I/O is integrated with formatted input and output.

Other. The operating system and utilities have been enhanced and freed of restrictions in many other ways too numerous to relate.

---

† UNIX is a Trademark of Bell Laboratories.

B. Hardware

The UNIX/32V operating system runs on a DEC VAX-11/780* with at least the following equipment:

memory: 256K bytes or more.

disk: RP06, RM03, or equivalent.

tape: any 9-track MASSBUS-compatible tape drive.

The following equipment is strongly recommended:

communications controller such as DZ11 or DL11.

full duplex 96-character ASCII terminals.

extra disk for system backup.

The system is normally distributed on 9-track tape. The minimum memory and disk space specified is enough to run and maintain UNIX/32V, and to keep all source on line. More memory will be needed to handle a large number of users, big data bases, diversified complements of devices, or large programs. The resident code occupies 40-55K bytes depending on configuration; system data also occupies 30-55K bytes.

C. Software

Most of the programs available as UNIX/32V commands are listed. Source code and printed manuals are distributed for all of the listed software except games. Almost all of the code is written in C. Commands are self-contained and do not require extra setup information, unless specifically noted as "interactive." Interactive programs can be made to run from a prepared script simply by redirecting input. Most programs intended for interactive use (e.g., the editor) allow for an escape to command level (the Shell). Most file processing commands can also go from standard input to standard output ("filters"). The piping facility of the Shell may be used to connect such filters directly to the input or output of other programs.

1. Basic Software

This includes the time-sharing operating system with utilities, and a compiler for the programming language C—enough software to write and run new applications and to maintain or modify UNIX/32V itself.

1.1. Operating System

□ UNIX — The basic resident code on which everything else depends. Supports the system calls, and maintains the file system. A general description of UNIX design philosophy and system facilities appeared in the Communications of the ACM, July, 1974. A more extensive survey is in the Bell System Technical Journal for July-August 1978. Capabilities include:

- ○ Reentrant code for user processes.
- ○ "Group" access permissions for cooperative projects, with overlapping memberships.
- ○ Alarm-clock timeouts.
- ○ Timer-interrupt sampling and interprocess monitoring for debugging and measurement.
- ○ Multiplexed I/O for machine-to-machine communication.

□ DEVICES — All I/O is logically synchronous. I/O devices are simply files in the file system. Normally, invisible buffering makes all physical record structure and device characteristics transparent and exploits the hardware's ability to do overlapped I/O. Unbuffered physical record I/O is available for unusual applications. Drivers for these devices are

*VAX is a Trademark of Digital Equipment Corporation.

available:
- O Asynchronous interfaces: DZ11, DL11. Support for most common ASCII terminals.
- O Automatic calling unit interface: DN11.
- O Printer/plotter: Versatek.
- O Magnetic tape: TE16.
- O Pack type disk: RP06, RM03; minimum-latency seek scheduling.
- O Physical memory of VAX-11, or mapped memory in resident system.
- O Null device.
- O Recipies are supplied to aid the construction of drivers for:
  - Asynchronous interface: DH11.
  - Synchronous interface: DU11.
  - DECtape: TC11.
  - Fixed head disk: RS11, RS03 and RS04.
  - Cartridge-type disk: RK05.
  - Phototypesetter: Graphic Systems System/1 through DR11C.

□ BOOT Procedures to get UNIX/32V started.

1.2. User Access Control

□ LOGIN Sign on as a new user.
- O Verify password and establish user's individual and group (project) identity.
- O Adapt to characteristics of terminal.
- O Establish working directory.
- O Announce presence of mail (from MAIL).
- O Publish message of the day.
- O Execute user-specified profile.
- O Start command interpreter or other initial program.

□ PASSWD Change a password.
- O User can change his own password.
- O Passwords are kept encrypted for security.

□ NEWGRP Change working group (project). Protects against unauthorized changes to projects.

1.3. Terminal Handling

□ TABS Set tab stops appropriately for specified terminal type.

□ STTY Set up options for optimal control of a terminal. In so far as they are deducible from the input, these options are set automatically by LOGIN.
- O Half vs. full duplex.
- O Carriage return+line feed vs. newline.
- O Interpretation of tabs.
- O Parity.
- O Mapping of upper case to lower.
- O Raw vs. edited input.
- O Delays for tabs, newlines and carriage returns.

1.4. File Manipulation

□ CAT Concatenate one or more files onto standard output. Particularly used for unadorned printing, for inserting data into a pipeline, and for buffering output that comes in dribs and drabs. Works on any file regardless of contents.

□ CP — Copy one file to another, or a set of files to a directory. Works on any file regardless of contents.

□ PR — Print files with title, date, and page number on every page.
- O Multicolumn output.
- O Parallel column merge of several files.

□ LPR — Off-line print. Spools arbitrary files to the line printer.

□ CMP — Compare two files and report if different.

□ TAIL — Print last *n* lines of input
- O May print last *n* characters, or from *n* lines or characters to end.

□ SPLIT — Split a large file into more manageable pieces. Occasionally necessary for editing (ED).

□ DD — Physical file format translator, for exchanging data with foreign systems, especially IBM 370's.

□ SUM — Sum the words of a file.

1.5. Manipulation of Directories and File Names

□ RM — Remove a file. Only the name goes away if any other names are linked to the file.
- O Step through a directory deleting files interactively.
- O Delete entire directory hierarchies.

□ LN — "Link" another name (alias) to an existing file.

□ MV — Move a file or files. Used for renaming files.

□ CHMOD — Change permissions on one or more files. Executable by files' owner.

□ CHOWN — Change owner of one or more files.

□ CHGRP — Change group (project) to which a file belongs.

□ MKDIR — Make a new directory.

□ RMDIR — Remove a directory.

□ CD — Change working directory.

□ FIND — Prowl the directory hierarchy finding every file that meets specified criteria.
- O Criteria include:
  - name matches a given pattern,
  - creation date in given range,
  - date of last use in given range,
  - given permissions,
  - given owner,
  - given special file characteristics,
  - boolean combinations of above.
- O Any directory may be considered to be the root.
- O Perform specified command on each file found.

1.6. Running of Programs

□ SH — The Shell, or command language interpreter.
- O Supply arguments to and run any executable program.
- O Redirect standard input, standard output, and standard error files.

- O Pipes: simultaneous execution with output of one process connected to the input of another.
- O Compose compound commands using:
  - if ... then ... else,
  - case switches,
  - while loops,
  - for loops over lists,
  - break, continue and exit,
  - parentheses for grouping.
- O Initiate background processes.
- O Perform Shell programs, i.e., command scripts with substitutable arguments.
- O Construct argument lists from all file names satisfying specified patterns.
- O Take special action on traps and interrupts.
- O User-settable search path for finding commands.
- O Executes user-settable profile upon login.
- O Optionally announces presence of mail as it arrives.
- O Provides variables and parameters with default setting.

□ TEST — Tests for use in Shell conditionals.
- O String comparison.
- O File nature and accessibility.
- O Boolean combinations of the above.

□ EXPR — String computations for calculating command arguments.
- O Integer arithmetic
- O Pattern matching

□ WAIT — Wait for termination of asynchronously running processes.

□ READ — Read a line from terminal, for interactive Shell procedure.

□ ECHO — Print remainder of command line. Useful for diagnostics or prompts in Shell programs, or for inserting data into a pipeline.

□ SLEEP — Suspend execution for a specified time.

□ NOHUP — Run a command immune to hanging up the terminal.

□ NICE — Run a command in low (or high) priority.

□ KILL — Terminate named processes.

□ CRON — Schedule regular actions at specified times.
- O Actions are arbitrary programs.
- O Times are conjunctions of month, day of month, day of week, hour and minute. Ranges are specifiable for each.

□ AT — Schedule a one-shot action for an arbitrary time.

□ TEE — Pass data between processes and divert a copy into one or more files.

1.7. Status Inquiries

□ LS — List the names of one, several, or all files in one or more directories.
- O Alphabetic or temporal sorting, up or down.
- O Optional information: size, owner, group, date last modified, date last accessed, permissions, i-node number.

□ FILE — Try to determine what kind of information is in a file by consulting the file system index and by reading the file itself.

- □ DATE — Print today's date and time. Has considerable knowledge of calendric and horological peculiarities.
  - O May set UNIX/32V's idea of date and time.
- □ DF — Report amount of free space on file system devices.
- □ DU — Print a summary of total space occupied by all files in a hierarchy.
- □ QUOT — Print summary of file space usage by user id.
- □ WHO — Tell who's on the system.
  - O List of presently logged in users, ports and times on.
  - O Optional history of all logins and logouts.
- □ PS — Report on active processes.
  - O List your own or everybody's processes.
  - O Tell what commands are being executed.
  - O Optional status information: state and scheduling info, priority, attached terminal, what it's waiting for, size.
- □ IOSTAT — Print statistics about system I/O activity.
- □ TTY — Print name of your terminal.
- □ PWD — Print name of your working directory.

1.8. Backup and Maintenance

- □ MOUNT — Attach a device containing a file system to the tree of directories. Protects against nonsense arrangements.
- □ UMOUNT — Remove the file system contained on a device from the tree of directories. Protects against removing a busy device.
- □ MKFS — Make a new file system on a device.
- □ MKNOD — Make an i-node (file system entry) for a special file. Special files are physical devices, virtual devices, physical memory, etc.
- □ TP
- □ TAR — Manage file archives on magnetic tape or DECtape. TAR is newer.
  - O Collect files into an archive.
  - O Update DECtape archive by date.
  - O Replace or delete DECtape files.
  - O Print table of contents.
  - O Retrieve from archive.
- □ DUMP — Dump the file system stored on a specified device, selectively by date, or indiscriminately.
- □ RESTOR — Restore a dumped file system, or selectively retrieve parts thereof.
- □ SU — Temporarily become the super user with all the rights and privileges thereof. Requires a password.
- □ DCHECK
- □ ICHECK
- □ NCHECK — Check consistency of file system.
  - O Print gross statistics: number of files, number of directories, number of special files, space used, space free.

- O Report duplicate use of space.
- O Retrieve lost space.
- O Report inaccessible files.
- O Check consistency of directories.
- O List names of all files.

□ CLRI — Peremptorily expunge a file and its space from a file system. Used to repair damaged file systems.

□ SYNC — Force all outstanding I/O on the system to completion. Used to shut down gracefully.

1.9. Accounting

The timing information on which the reports are based can be manually cleared or shut off completely.

□ AC — Publish cumulative connect time report.
- O Connect time by user or by day.
- O For all users or for selected users.

□ SA — Publish Shell accounting report. Gives usage information on each command executed.
- O Number of times used.
- O Total system time, user time and elapsed time.
- O Optional averages and percentages.
- O Sorting on various fields.

1.10. Communication

□ MAIL — Mail a message to one or more users. Also used to read and dispose of incoming mail. The presence of mail is announced by LOGIN and optionally by SH.
- O Each message can be disposed of individually.
- O Messages can be saved in files or forwarded.

□ CALENDAR Automatic reminder service for events of today and tomorrow.

□ WRITE — Establish direct terminal communication with another user.

□ WALL — Write to all users.

□ MESG — Inhibit receipt of messages from WRITE and WALL.

□ CU — Call up another time-sharing system.
- O Transparent interface to remote machine.
- O File transmission.
- O Take remote input from local file or put remote output into local file.
- O Remote system need not be UNIX/32V.

□ UUCP — UNIX to UNIX copy.
- O Automatic queuing until line becomes available and remote machine is up.
- O Copy between two remote machines.
- O Differences, mail, etc., between two machines.

1.11. Basic Program Development Tools

Some of these utilities are used as integral parts of the higher level languages described in section 2.

□ AR — Maintain archives and libraries. Combines several files into one for housekeeping efficiency.
- O Create new archive.

- O Update archive by date.
- O Replace or delete files.
- O Print table of contents.
- O Retrieve from archive.

□ AS — Assembler.

- O Creates object program consisting of
  - code, normally read-only and sharable,
  - initialized data or read-write code,
  - uninitialized data.
- O Relocatable object code is directly executable without further transformation.
- O Object code normally includes a symbol table.
- O "Conditional jump" instructions become branches or branches plus jumps depending on distance.

□ Library — The basic run-time library. These routines are used freely by all software.

- O Buffered character-by-character I/O.
- O Formatted input and output conversion (SCANF and PRINTF) for standard input and output, files, in-memory conversion.
- O Storage allocator.
- O Time conversions.
- O Number conversions.
- O Password encryption.
- O Quicksort.
- O Random number generator.
- O Mathematical function library, including trigonometric functions and inverses, exponential, logarithm, square root, bessel functions.

□ ADB — Interactive debugger.

- O Postmortem dumping.
- O Examination of arbitrary files, with no limit on size.
- O Interactive breakpoint debugging with the debugger as a separate process.
- O Symbolic reference to local and global variables.
- O Stack trace for C programs.
- O Output formats:
  - 1-, 2-, or 4-byte integers in octal, decimal, or hex
  - single and double floating point
  - character and string
  - disassembled machine instructions
- O Patching.
- O Searching for integer, character, or floating patterns.

□ OD — Dump any file. Output options include any combination of octal or decimal or hex by words, octal by bytes, ASCII, opcodes, hexadecimal.

- O Range of dumping is controllable.

□ LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.

- O Resulting code is sharable by default.

□ LORDER — Places object file names in proper order for loading, so that files depending on others come after them.

□ NM — Print the namelist (symbol table) of an object program. Provides control over the style and order of names that are printed.

□ SIZE — Report the memory requirements of one or more object files.

□ STRIP Remove the relocation and symbol table information from an object file to save space.

□ TIME Run a command and report timing information on it.

□ PROF Construct a profile of time spent per routine from statistics gathered by time-sampling the execution of a program.
O Subroutine call frequency and average times for C programs.

□ MAKE Controls creation of large programs. Uses a control file specifying source file dependencies to make new version; uses time last changed to deduce minimum amount of work necessary.
O Knows about CC, YACC, LEX, etc.

1.12. UNIX/32V Programmer's Manual

□ Manual Machine-readable version of the UNIX/32V Programmer's Manual.
O System overview.
O All commands.
O All system calls.
O All subroutines in C and assembler libraries.
O All devices and other special files.
O Formats of file system and kinds of files known to system software.
O Boot and maintenance procedures.

□ MAN Print specified manual section on your terminal.

1.13. Computer-Aided Instruction

□ LEARN A program for interpreting CAI scripts, plus scripts for learning about UNIX/32V by using it.
O Scripts for basic files and commands, editor, advanced files and commands, EQN, MS macros, C programming language.

2. Languages

2.1. The C Language

□ CC Compile and/or link edit programs in the C language. The UNIX/32V operating system, most of the subsystems and C itself are written in C. For a full description of C, read *The C Programming Language,* Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.
O General purpose language designed for structured programming.
O Data types include character, integer, float, double, pointers to all types, functions returning above types, arrays of all types, structures and unions of all types.
O Operations intended to give machine-independent control of full machine facility, including to-memory operations and pointer arithmetic.
O Macro preprocessor for parameterized code and inclusion of standard files.
O All procedures recursive, with parameters by value.
O Machine-independent pointer manipulation.
O Object code uses full addressing capability of the VAX-11.
O Runtime library gives access to all system facilities.
O Definable data types.
O Block structure

□ LINT Verifier for C programs. Reports questionable or nonportable usage such as:
Mismatched data declarations and procedure interfaces.
Nonportable type conversions.

Unused variables, unreachable code, no-effect operations.
Mistyped pointers.
Obsolete syntax.
- O Full cross-module checking of separately compiled programs.

□ CB — A beautifier for C programs. Does proper indentation and placement of braces.

2.2. Fortran

□ F77 — A full compiler for ANSI Standard Fortran 77.
- O Compatible with C and supporting tools at object level.
- O Optional source compatibility with Fortran 66.
- O Free format source.
- O Optional subscript-range checking, detection of uninitialized variables.
- O All widths of arithmetic: 2- and 4-byte integer; 4- and 8-byte real; 8- and 16-byte complex.

□ RATFOR — Ratfor adds rational control structure à la C to Fortran.
- O Compound statements.
- O If-else, do, for, while, repeat-until, break, next statements.
- O Symbolic constants.
- O File insertion.
- O Free format source
- O Translation of relationals like >, >=.
- O Produces genuine Fortran to carry away.
- O May be used with F77.

□ STRUCT — Converts ordinary ugly Fortran into structured Fortran (i.e., Ratfor), using statement grouping, if-else, while, for, repeat-until.

2.3. Other Algorithmic Languages

□ DC — Interactive programmable desk calculator. Has named storage locations as well as conventional stack for holding integers or programs.
- O Unlimited precision decimal arithmetic.
- O Appropriate treatment of decimal fractions.
- O Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.
- O Reverse Polish operators:
  + - * /
  remainder, power, square root,
  load, store, duplicate, clear,
  print, enter program text, execute.

□ BC — A C-like interactive interface to the desk calculator DC.
- O All the capabilities of DC with a high-level syntax.
- O Arrays and recursive functions.
- O Immediate evaluation of expressions and evaluation of functions upon call.
- O Arbitrary precision elementary functions: exp, sin, cos, atan.
- O Go-to-less programming.

2.4. Macroprocessing

□ M4 — A general purpose macroprocessor.
- O Stream-oriented, recognizes macros anywhere in text.
- O Syntax fits with functional syntax of most higher-level languages.

○ Can evaluate integer arithmetic expressions.

2.5. Compiler-compilers

□ YACC — An LR(1)-based compiler writing system. During execution of resulting parsers, arbitrary C functions may be called to do code generation or semantic actions.
○ BNF syntax specifications.
○ Precedence relations.
○ Accepts formally ambiguous grammars with non-BNF resolution rules.

□ LEX — Generator of lexical analyzers. Arbitrary C functions may be called upon isolation of each lexical token.
○ Full regular expression, plus left and right context dependence.
○ Resulting lexical analysers interface cleanly with YACC parsers.

3. Text Processing

3.1. Document Preparation

□ ED — Interactive context editor. Random access to all lines of a file.
○ Find lines by number or pattern. Patterns may include: specified characters, don't care characters, choices among characters, repetitions of these constructs, beginning of line, end of line.
○ Add, delete, change, copy, move or join lines.
○ Permute or split contents of a line.
○ Replace one or all instances of a pattern within a line.
○ Combine or split files.
○ Escape to Shell (command language) during editing.
○ Do any of above operations on every pattern-selected line in a given range.
○ Optional encryption for extra security.

□ PTX — Make a permuted (key word in context) index.

□ SPELL — Look for spelling errors by comparing each word in a document against a word list.
○ 25,000-word list includes proper names.
○ Handles common prefixes and suffixes.
○ Collects words to help tailor local spelling lists.

□ LOOK — Search for words in dictionary that begin with specified prefix.

□ CRYPT — Encrypt and decrypt files for security.

3.2. Document Formatting

□ TROFF

□ NROFF — Advanced typesetting. TROFF drives a Graphic Systems phototypesetter; NROFF drives ascii terminals of all types. This summary was typeset using TROFF. TROFF and NROFF are capable of elaborate feats of formatting, when appropriately programmed. TROFF and NROFF accept the same input language.
○ Completely definable page format keyed to dynamically planted "interrupts" at specified lines.
○ Maintains several separately definable typesetting environments (e.g., one for body text, one for footnotes, and one for unusually elaborate headings).
○ Arbitrary number of output pools can be combined at will.
○ Macros with substitutable arguments, and macros invocable in mid-line.

- O Computation and printing of numerical quantities.
- O Conditional execution of macros.
- O Tabular layout facility.
- O Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.
- O Access to character-width computation for unusually difficult layout problems.
- O Overstrikes, built-up brackets, horizontal and vertical line drawing.
- O Dynamic relative or absolute positioning and size selection, globally or at the character level.
- O Can exploit the characteristics of the terminal being used, for approximating special characters, reverse motions, proportional spacing, etc.

The Graphic Systems typesetter has a vocabulary of several 102-character fonts (4 simultaneously) in 15 sizes. TROFF provides terminal output for rough sampling of the product.

NROFF will produce multicolumn output on terminals capable of reverse line feed, or through the postprocessor COL.

High programming skill is required to exploit the formatting capabilities of TROFF and NROFF, although unskilled personnel can easily be trained to enter documents according to canned formats such as those provided by MS, below. TROFF and EQN are essentially identical to NROFF and NEQN so it is usually possible to define interchangeable formats to produce approximate proof copy on terminals before actual typesetting. The preprocessors MS, TBL, and REFER are fully compatible with TROFF and NROFF.

□ MS — A standardized manuscript layout package for use with NROFF/TROFF. This document was formatted with MS.

- O Page numbers and draft dates.
- O Automatically numbered subheads.
- O Footnotes.
- O Single or double column.
- O Paragraphing, display and indentation.
- O Numbered equations.

□ EQN — A mathematical typesetting preprocessor for TROFF. Translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions. Formulas are written in a style like this:

```
sigma sup 2 "=" 1 over N sum from i=1 to N ( x sub i - x bar ) sup 2
```

which produces:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

- O Automatic calculation of size changes for subscripts, sub-subscripts, etc.
- O Full vocabulary of Greek letters and special symbols, such as 'gamma', 'GAMMA', 'integral'.
- O Automatic calculation of large bracket sizes.
- O Vertical "piling" of formulae for matrices, conditional alternatives, etc.
- O Integrals, sums, etc., with arbitrarily complex limits.
- O Diacriticals: dots, double dots, hats, bars, etc.
- O Easily learned by nonprogrammers and mathematical typists.

□ NEQN — A version of EQN for NROFF; accepts the same input language. Prepares formulas for display on any terminal that NROFF knows about, for example, those based on Diablo printing mechanism.

- O Same facilities as EQN within graphical capability of terminal.

□ TBL — A preprocessor for NROFF/TROFF that translates simple descriptions of table layouts and contents into detailed typesetting instructions.
- O Computes column widths.
- O Handles left- and right-justified columns, centered columns and decimal-point alignment.
- O Places column titles.
- O Table entries can be text, which is adjusted to fit.
- O Can box all or parts of table.

□ REFER — Fills in bibliographic citations in a document from a data base (not supplied).
- O References may be printed in any style, as they occur or collected at the end.
- O May be numbered sequentially, by name of author, etc.

□ TC — Simulate Graphic Systems typesetter on Tektronix 4014 scope. Useful for checking TROFF page layout before typesetting.

□ COL — Canonicalize files with reverse line feeds for one-pass printing.

□ DEROFF — Remove all TROFF commands from input.

□ CHECKEQ — Check document for possible errors in EQN usage.

4. Information Handling

□ SORT — Sort or merge ASCII files line-by-line. No limit on input size.
- O Sort up or down.
- O Sort lexicographically or on numeric key.
- O Multiple keys located by delimiters or by character position.
- O May sort upper case together with lower into dictionary order.
- O Optionally suppress duplicate data.

□ TSORT — Topological sort — converts a partial order into a total order.

□ UNIQ — Collapse successive duplicate lines in a file into one line.
- O Publish lines that were originally unique, duplicated, or both.
- O May give redundancy count for each line.

□ TR — Do one-to-one character translation according to an arbitrary code.
- O May coalesce selected repeated characters.
- O May delete selected characters.

□ DIFF — Report line changes, additions and deletions necessary to bring two files into agreement.
- O May produce an editor script to convert one file into another.
- O A variant compares two new versions against one old one.

□ COMM — Identify common lines in two sorted files. Output in up to 3 columns shows lines present in first file only, present in both, and/or present in second only.

□ JOIN — Combine two files by joining records that have identical keys.

□ GREP — Print all lines in a file that satisfy a pattern as used in the editor ED.
- O May print all lines that fail to match.
- O May print count of hits.
- O May print first hit in each file.

□ LOOK — Binary search in sorted file for lines with specified prefix.

□ WC — Count the lines, "words" (blank-separated strings) and characters in a file.

□ SED — Stream-oriented version of ED. Can perform a sequence of editing operations on each line of an input stream of unbounded length.

- O Lines may be selected by address or range of addresses.
- O Control flow and conditional testing.
- O Multiple output streams.
- O Multi-line capability.

□ AWK Pattern scanning and processing language. Searches input for patterns, and performs actions on each line of input that satisfies the pattern.
- O Patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these.
- O Data treated as string or numeric as appropriate.
- O Can break input into fields; fields are variables.
- O Variables and arrays (with non-numeric subscripts).
- O Full set of arithmetic operators and control flow.
- O Multiple output streams to files and pipes.
- O Output can be formatted as desired.
- O Multi-line capabilities.

5. Graphics

The programs in this section are predominantly intended for use with Tektronix 4014 storage scopes.

□ GRAPH Prepares a graph of a set of input numbers.
- O Input scaled to fit standard plotting area.
- O Abscissae may be supplied automatically.
- O Graph may be labeled.
- O Control over grid style, line style, graph orientation, etc.

□ SPLINE Provides a smooth curve through a set of points intended for GRAPH.

□ PLOT A set of filters for printing graphs produced by GRAPH and other programs on various terminals. Filters provided for 4014, DASI terminals, Versatec printer/plotter.

6. Novelties, Games, and Things That Didn't Fit Anywhere Else

□ BACKGAMMON
A player of modest accomplishment.

□ BCD Converts ascii to card-image form.

□ CAL Print a calendar of specified month and year.

□ CHING The *I Ching.* Place your own interpretation on the output.

□ FORTUNE Presents a random fortune cookie on each invocation. Limited jar of cookies included.

□ UNITS Convert amounts between different scales of measurement. Knows hundreds of units. For example, how many km/sec is a parsec/megayear?

□ ARITHMETIC
Speed and accuracy test for number facts.

□ QUIZ Test your knowledge of Shakespeare, Presidents, capitals, etc.

□ WUMP Hunt the wumpus, thrilling search in a dangerous cave.

□ HANGMAN Word-guessing game. Uses a dictionary supplied with SPELL.

□ FISH Children's card-guessing game.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on the UNIX† system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

† UNIX is a trademark of AT&T Bell Laboratories.

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if `prog` uses `getchar`, then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using >: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()  /* ccstrip: strip non-graphic characters */
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE    *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type

name, like `int`, not a structure tag.

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (`"r"`), write (`"w"`), or append (`"a"`).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. `getc` and `putc` return `EOF` on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)    /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

The function `fprintf` is identical to `printf`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a subprocess. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns `EOF` at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns `NULL` at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` "pushes back" the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor.* Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor

0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512 /* best size for PDP-11 UNIX */

main()  /* copy input to output */
{
    char    buf[BUFSIZE];
    int n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define CMASK   0377    /* for making char's > 0 */

getchar()   /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with `0377` to ensure that it is positive; otherwise sign extension may make it negative. (The constant `0377` is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#define CMASK   0377    /* for making char's > 0 */
#define BUFSIZE 512

getchar()   /* buffered version */
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int  n = 0;

    if (n == 0) {   /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)    /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int f1, f2, n;
    char    buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2)   /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0);  /* get to pos */
    return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of −1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual,* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember than `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is

zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);    /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the `command` and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's `system` routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `forks` to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `execl`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ     0
#define WRITE    1
#define tst(a, b)   (mode == READ ? (b) : (a))
static  int popen_pid;

popen(cmd, mode)
char    *cmd;
int mode;
{
    int p[2];

    if (pipe(p) < 0)
        return(NULL);
    if ((popen_pid = fork()) == 0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", cmd, 0);
        _exit(1);   /* disaster has occurred if we get here */
    }
    if (popen_pid == -1)
        return(NULL);
    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}
```

The sequence of `close`s in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit

tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)  /* close pipe fd */
int fd;
{
    register r, (*hstat)(), (*istat)(), (*qstat)();
    int  status;
    extern int popen_pid;

    close(fd);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
    while ((r = wait(&status)) != popen_pid && r != -1);
    if (r == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);
    return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */

    exit(0);
}

onintr()
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
    int (*istat)(), onintr();

    istat = signal(SIGINT, SIG_IGN);    /* save original status */
    setjmp(sjbuf);  /* save current stack position */
    if (istat != SIG_IGN)
        signal(SIGINT, onintr);

    /* main processing loop */
}

onintr()
{
    printf("\nInterrupt\n");
    longjmp(sjbuf); /* return to saved state */
}
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
    if (intflag)
        /* EOF caused by interrupt */
    else
        /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like

this:

```
if (fork() == 0)
    execl(...);
signal(SIGINT, SIG_IGN);    /* ignore interrupts */
wait(&status);  /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s)   /* run command string s */
char *s;
{
    int status, pid, w;
    register int (*istat)(), (*qstat)();

    if ((pid = fork()) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
    }
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

References

[1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual,* Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF is actually -1, and is the value returned by the read routines on end-of-file or error.

NULL is a notation for the null pointer, returned by pointer-valued functions to indicate an error

FILE expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.

getc, getchar, putc, putchar, feof, ferror, fileno
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

2. Calls

`FILE *fopen(filename, type) char *filename, *type;`
opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

`FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;`
The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

`int getc(ioptr) FILE *ioptr;`
returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer `EOF` is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

`int fgetc(ioptr) FILE *ioptr;`
acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`
`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but `EOF` is returned on error.

`fputc(c, ioptr) FILE *ioptr;`
acts like `putc` but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`
The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`
Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`
terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(ioptr) FILE *ioptr;`
returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`
is identical to `getc(stdin)`.

`putchar(c);`
is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`
reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or `NULL` if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`
writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`
The argument character `c` is pushed back on the input stream named by `ioptr`. Only one character may be pushed back.

```
printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...)char *s, *format;
```
`printf` writes on the standard output. `fprintf` writes on the named output stream. `sprintf` puts characters in the character array (string) named by `s`. The specifications are as described in section `printf`(3) of the *UNIX Programmer's Manual.*

```
scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;
```

`scanf` reads from the standard input. `fscanf` reads from the named input stream. `sscanf` reads from the character string supplied as `s`. `scanf` reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string `format`, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

`scanf` returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

```
fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
```

reads `nitems` of data beginning at `ptr` from file `ioptr`. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the `fopen` call.

```
fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
```

Like `fread`, but in the other direction.

```
rewind(ioptr) FILE *ioptr;
```

rewinds the stream named by `ioptr`. It is not very useful except on input, since a rewound output file is still open only for output.

```
system(string) char *string;
```

The `string` is executed by the shell as if typed at the terminal.

```
getw(ioptr) FILE *ioptr;
```

returns the next word from the input stream named by `ioptr`. `EOF` is returned on end-of-file or error, but since this a perfectly good integer `feof` and `ferror` should be used. A "word" is 16 bits on the PDP-11.

```
putw(w, ioptr) FILE *ioptr;
```

writes the integer `w` on the named output stream.

```
setbuf(ioptr, buf) FILE *ioptr; char *buf;
```

`setbuf` may be used after a stream has been opened but before I/O has started. If `buf` is `NULL`, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char    buf[BUFSIZ];
```

```
fileno(ioptr) FILE *ioptr;
```

returns the integer file descriptor associated with the file.

```
fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
```

The location of the next byte in the stream named by `ioptr` is adjusted. `offset` is a long integer. If `ptrname` is 0, the offset is measured from the beginning of the file; if `ptrname` is 1, the offset is measured from the current read or write pointer; if `ptrname` is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the ptrname must be 0).

```
long ftell(ioptr) FILE *ioptr;
```

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`
The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`
allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`
allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available .

`cfree(ptr) char *ptr;`
Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

UNIX Implementation

K. Thompson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes

† UNIX is a trademark of AT&T Bell Laboratories.

loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

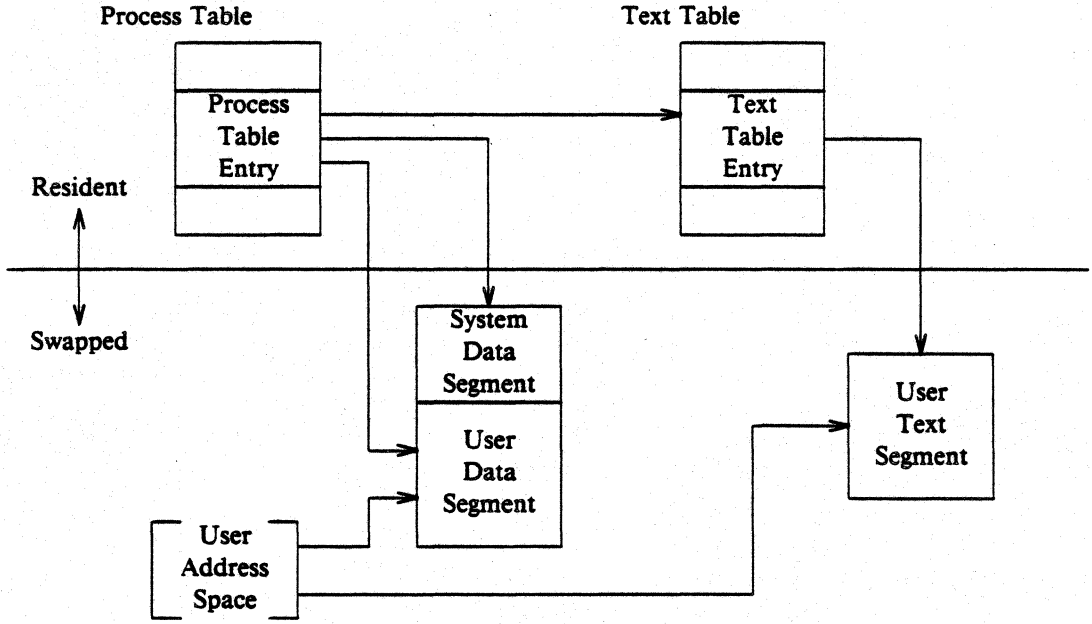

Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[2]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The

character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 3.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the

so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was

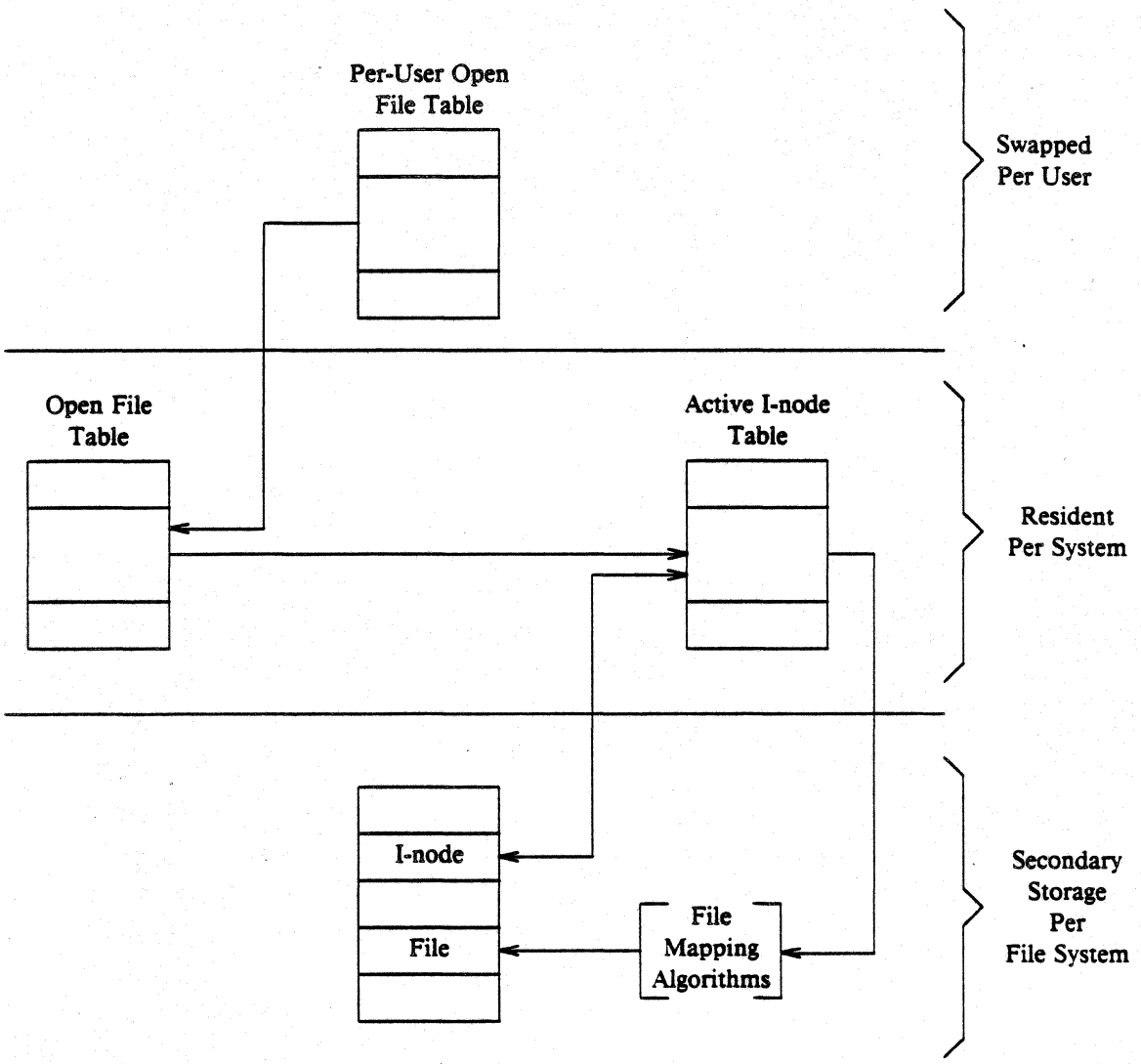

Fig. 2—File system data structure.

the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2. In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O

pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe**. Implementation of **pipes** consists of implied **seeks** before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user–a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[3] Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[2]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices*, vol. 12, no. 4, pp. 40-50, April 1977.
2. E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, pp. 43-112, Academic Press, New York, 1968.
3. This issue, D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.

The UNIX I/O System

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character.* The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u.* This table is indexed by the file descriptor returned by the *open* or *creat,* and is accessed during a *read, write,* or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat.* This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a

†UNIX is a Trademark of Bell Laboratories.

pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close.*

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on,* that

u.u_base refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write,* except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: *(*p) (dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2].*

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc.* A queue header has the structure

```
struct {
        int     c_cc; /* character count */
        char    *c_cf;      /* first character */
        char    *c_cl;      /* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority.*

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the

waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup*, (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf*. Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block

until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread,* but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERRORThis bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTED
This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes

down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRI
This bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE. Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

The Programming Language EFL

Stuart I. Feldman

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Fortran
Preprocessors
Ratfor

ABSTRACT

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. The EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment. EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. The current version of the EFL compiler is written in portable C.

1. INTRODUCTION

1.1. Purpose

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

1.2. History

EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. A. D. Hall designed the initial version of the language and wrote a preliminary version of a compiler. I extended and modified the language and wrote a full compiler (in C) for it. The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions. To achieve this goal, a sizable two-pass translator is needed.

1.3. Notation

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

item

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

2. LEXICAL FORM

2.1. Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + - * /
	= < > & ~ | $

Letter case (upper or lower) is ignored except within strings, so '**a**' and 'A' are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ('!') may be used in place of a tilde ('~'). Square brackets ('[' and ']') may be used in place of braces ('{' and '}').

2.2. Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

2.2.1. White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

2.2.2. Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

2.2.3. Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

2.2.4. Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

2.2.5. Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

2.3. Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

2.3.1. Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

2.3.2. Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

```
'hello there'
"ain't misbehavin"
```

2.3.3. Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

2.3.4. Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

2.3.5. Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

2.3.6. Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  -  *  /  **
<  <=  >  >=  ==  ~=
&&  ||  &  |
+=  -=  /=  **=
&&=  ||=  &=  |=
->  .  $
```

A dot ('.') is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see the Atavisms section) in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.*, **.lt.**).

2.4. Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

n += 1

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

3. PROGRAM FORM

3.1. Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

3.2. Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in Section 8.

3.3. Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations there are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See Section 7.2). An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#   block 0
procedure george
real x
x = 2
...
if(x > 2)
                {           # new block
                integer x   # a different variable
                do x = 1,7
                            write(,x)
                ...
                }           # end of block
end             # end of procedure, return to block 0
```

3.4. Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in Section 10. The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statements and finishes with an **end** statement; these are discussed in Section 8. Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

3.5. Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        . . .
error:  fatal("bad input")
```

4. DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

4.1. Basic Types

The basic types are

logical
integer
field(*m*:*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values true and false. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A 'real' quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

4.2. Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
-94
+6
0

A long real ('double precision') constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ('single precision') constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
-.4
7.9e-6 (= 7.9×10^{-6})
14e9 (= 1.4×10^{10})

The following are valid **long real** constants

7.9d-6 (= 7.9×10^{-6})
5d3

A character constant is a quoted string.

4.3. Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

4.3.1. Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

4.3.2. Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

4.3.3. Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

4.4. Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

4.5. Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

5. EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(expression)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in sections 5.3 and 5.4.

```
-> .
**
* /    unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=   &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

5.1. Primaries

Primaries are the basic elements of expressions, as follows:

5.1.1. Constants

Constants are described in Section 4.2.

5.1.2. Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may only appear as procedure arguments and in input/output lists.

5.1.3. Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6,-3,4)
```

5.1.4. Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

5.1.5. Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see Section 8.5), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See Chapter 8 for details.

5.1.6. Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See Section 7.7.

5.1.7. Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = **5**
long real(5) = **5.0d0**
complex(5,3) = 5+3*i*

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

5.1.8. Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable x in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

5.2. Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

5.3. Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1. Arithmetic

Unary + has no effect. A unary – yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator — subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

5.3.2. Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

5.4. Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

5.4.1. Arithmetic

The binary arithmetic operators are

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

5.4.2. Logical

The two binary logical operations in EFL, **and** and **or**, are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise the expression has the value of **b**. The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise the expression has the value of **b**. The other forms of the operators (**&** for **and** and **|** for **or**) do

not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

5.4.3. Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	<	less than
<=	≤	less than or equal to
==	=	equal to
~=	≠	not equal to
>	>	greater than
>=	≥	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

5.4.4. Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

5.5. Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside -> structurename

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) -> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

5.6. Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

5.7. Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

6.1. Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two form:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

6.2. Attributes

6.2.1. Basic Types

The following are basic types in declarations

logical
integer
field(*m*:*n*)
character(*k*)
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

6.2.2. Arrays

The dimensionality may be declared by an **array** attribute

array(*b*$_1$,...,*b*$_n$)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper

bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper* − *lower* + 1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)
```

6.2.3. Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three xx's and a character string.

6.2.4. Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

6.2.5. Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

6.2.6. External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external *name*

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

6.3. Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

6.4. The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial *var* = *val*

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

7. EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements — otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

7.1. Expression Statements

7.1.1. Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see Section 7.7) resemble procedure invocations but do not yield a value. If an error occurs the program stops.

7.1.2. Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

7.2. Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i     # this variable is unknown outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

7.3. Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

7.3.1. If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

7.3.2. If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □ **else** □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-elsed **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

7.3.3. Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression*) □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case *constant* :

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not 'fall through' to the next case.

7.4. Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

7.4.1. While Statement

This construct has the form

while (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

7.5. For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , □ *logical-expression* , □ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see Section 7.6.3), this construct is equivalent to

```
initial-statement
while ( logical-expression )
   {
   body-statement
   iteration-statement
   }
```

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
      n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
    ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

7.5.1. Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

7.5.2. Repeat...Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise control returns to the *statement.* Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

7.5.3. Do Loops

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable* = *expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable = expression-1 , variable <= t2 , variable += t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

7.6. Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

7.6.1. Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
    {
    case 1:
        error(7)

    case 2:
        k = 2
        goto case 4

    case 3:
        k = 5
        goto case 4

    case 4:
        fixup(k)
        goto default

    default:
        prmsg("ouch")
    }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

7.6.2. Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
    {
    do a computation
    if( finished )
        break
    }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

7.6.3. Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

7.6.4. Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

7.7. Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

7.7.1. Input/Output Units

Each I/O statement refers to a 'unit', identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

7.7.2. Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* , *binary-output-list*)
readbin(*unit* , *binary-input-list*)

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

7.7.3. Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* , *formatted-output-list*)
read(*unit* , *formatted-input-list*)

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

7.7.4. Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

7.7.5. Formats

The following are permissible *format-specifiers.* The quantities w, d, and k must be integer constant expressions.

i(w)	integer with w digits
f(w,d)	floating point number of w characters, d of them to the right of the decimal point.
e(w,d)	floating point number of w characters, d of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(w)	logical field of width w characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(w)	character string of width w
s(k)	skip k lines
x(k)	skip k spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

7.7.6. Manipulation statements

The three input/output statements

backspace(*unit*)
rewind(*unit*)
endfile(*unit*)

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

8. PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

8.1. Procedure Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* ( )
attributes **procedure** *procedurename* (*name*)

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

8.2. End Statement

Each procedure terminates with a statement

end

8.3. Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

8.4. Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

8.5. Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

8.5.1. Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real.** Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, -3.20)
max(i, z)
```

8.5.2. Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

8.5.3. Elementary Functions

The following generic functions take arguments of **real, long real,** or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function** (e^x).
log	**natural (base** e**) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function** ($\sqrt{x}$).

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

8.5.4. Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y) \mid x \mid$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

9. ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

9.1. Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ('**%**') is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines

constitute a continued Fortran statement, they should be enclosed in braces.

9.2. Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

9.3. Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

9.4. Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

9.5. Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a–h, o–z) real
implicit (i–n) integer
```

9.6. Computed goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto (*label*), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

9.7. Go To Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

```
go to xyz
```

9.8. Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see Section 10.2) which forces the compiler to recognize the forms in the second column below:

<	.lt.
<=	.le.
>	.gt.
>=	.ge.
==	.eq.
~=	.ne.
&	.and.
|	.or.
&&	.andand.
||	.oror.
~	.not.
true	.true.
false	.false.

In this mode, no structure element may be named **lt**, **le**, etc. The readable forms in the left column are always recognized.

9.9. Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

9.10. Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

9.11. Equivalence

A statement of the form

equivalence $v_1, v_2, \ldots, v_n$

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12. Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	Argument Type	Result Type
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

10. COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option *opt*

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

10.1. Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

10.2. Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

10.3. Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

10.4. Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (&) in the first column of the continued lines instead.

10.5. Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	Type
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

10.6. Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	Size Option	Alignment Option
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

10.7. Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

10.8. Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **proc-header** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11. EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.1. File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
      write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.2. Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
            {
            c(i,j) = 0
            do k = 1,n
                        c(i,j) += a(i,k) * b(k,j)
            }
end
```

11.3. Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST          0
define NOTFOUND  -1

integer procedure val(list, first, x)

#   list is an array of structures.
#   Each structure contains a thread index value, an x, and a y value.
struct
            {
            integer nextindex
            integer x, y
            } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x , p = list(p).nextindex)
            if(list(p).x == x)
                        return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x >

x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex.**

11.4. Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
    print its value
otherwise
    print a left parenthesis
    print the left node
    print the operator
    print the right node
    print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)  # print out an expression tree

integer first                    # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100) # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODE     tree(currentnode)
define STACK    stackframe(stackdepth)

#  nextstate values
define DOWN     1
define LEFT     2
define RIGHT    3

#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first
```

```
while( stackdepth > 0 )
    {
    currentnode = STACK.nodep
    select(STACK.nextstate)
        {
        case DOWN:
            if(NODE.op == " ")  # a leaf
                {
                outval( NODE.val )
                stackdepth -= 1
                }
            else  {  # a binary operator node
                outch( "(" )
                STACK.nextstate = LEFT
                stackdepth += 1
                STACK.nextstate = DOWN
                STACK.nodep = NODE.leftp
                }

        case LEFT:
            outch( NODE.op )
            STACK.nextstate = RIGHT
            stackdepth += 1
            STACK.nextstate = DOWN
            STACK.nodep = NODE.rightp

        case RIGHT:
            outch( ")" )
            stackdepth -= 1
        }
    }
end
```

12. PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

12.1. Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1. Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a**.

12.1.2. Character String Comparisons

The function **efl cmc** is invoked to determine the order of two character strings. The declaration is

```
integer function efl cmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb**. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

13. ACKNOWLEDGMENTS

A. D. Hall originated the EFL language and wrote the first compiler for it; he also gave inestimable aid when I took up the project. B. W. Kernighan and W. S. Brown made a number of useful suggestions about the language and about this report. N. L. Schryer has acted as willing, cheerful, and severe first user and helpful critic of each new version and facility. J. L. Blue, L. C. Kaufman, and D. D. Warner made very useful contributions by making serious use of the compiler, and noting and tolerating its misbehaviors.

14. REFERENCE

1. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran", Bell Laboratories Computing Science Technical Report #55

APPENDIX A. Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the Atavisms section are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

APPENDIX B. COMPILER

B.1. Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers. Versions of this compiler run under the and UNIX† operating systems.

B.2. Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

B.3. Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (Section 11.2):

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
         do  2 j = 1, p
            c(i, j) = 0
            do  1 k = 1, n
               c(i, j) = c(i, j)+a(i, k)*b(k, j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk (Section 11.4):

† UNIX is a trademark of AT&T Bell Laboratories.

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
    1 if (stapth .le. 0) goto 9
        curode = staame(2, stapth)
        goto 7
    2     if (tree(1, curode) .ne. const1(1)) goto 3
            call outval(tree1(4, curode))
c a leaf
            stapth = stapth-1
            goto 4
    3       call outch(1h()
c a binary operator node
            staame(1, stapth) = 2
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(2, curode)
    4     goto 8
    5     call outch(tree(1, curode))
          staame(1, stapth) = 3
          stapth = stapth+1
          staame(1, stapth) = 1
          staame(2, stapth) = tree(3, curode)
          goto 8
    6     call outch(1h))
          stapth = stapth-1
          goto 8
    7     if (staame(1, stapth) .eq. 3) goto 6
          if (staame(1, stapth) .eq. 2) goto 5
          if (staame(1, stapth) .eq. 1) goto 2
    8   continue
        goto 1
    9 continue
      end
```

APPENDIX C. CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names),

but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

C.1. External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

C.2. Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

C.3. Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

C.4. Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

C.5. Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Berkeley FP User's Manual, Rev. 4.1

by

Scott Baden

ABSTRACT

This manual describes the Berkeley implementation of Backus' Functional Programming Language, FP. Since this implementation differs from Backus' original description of the language, those familiar with the literature will need to read about the system commands and the local modifications.

May 10, 1986

Table of Contents

1. Background

FP stands for a *Functional Programming* language. Functional programs deal with *functions* instead of *values*. There is no explicit representation of state, there are no assignment statments, and hence, no variables. Owing to the lack of state, FP functions are free from side-effects; so we say the FP is *applicative*.

All functions take one argument and they are evaluated using the single FP operation, *application* (the colon ':' is the apply operator). For example, we read +:<3 4> as "apply the function '+' to its argument <3 4>".

Functional programs express a functional-level combination of their components instead of describing state changes using value-oriented expressions. For example, we write the function returning the *sin* of the *cos* of its input, i.e., $sin(cos(x))$, as: sin@cos. This is a *functional expression*, consisting of the single *combining form* called *compose* ('@' is the compose operator) and its *functional arguments sin* and *cos*.

All combining forms take functions as arguments and return functions as results; functions may either be applied, e.g., sin@cos:3, or used as a functional argument in another functional expression, e.g., *tan @ sin @ cos*.

As we have seen, FP's combining forms allow us to express control abstractions without the use of variables. The *apply to all* functional form (&) is another case in point. The function '& exp' exponentiates all the elements of its argument:

$$\&exp : \langle 1.0\ 2.0\rangle \equiv \langle 2.718\ 7.389\rangle \tag{1.1}$$

In (1.1) there are no induction variables, nor a loop bounds specification. Moreover, the code is useful for any size argument, so long as the sub-elements of its argument conform to the domain of the *exp* function.

We must change our view of the programming process to adapt to the functional style. Instead of writing down a set of steps that manipulate and assign values, we compose functional expressions using the higher-level functional forms. For example, the function that adds a scalar to all elements of a vector will be written in two steps. First, the function that distributes the scalar amongst each element of the vector:

$$distl : \langle 3\ \langle 4\ 6\rangle\rangle \equiv \langle\langle 3\ 4\rangle\ \langle 3\ 6\rangle\rangle \tag{1.2}$$

Next, the function that adds the pairs of elements that make up a sequence:

$$\&+ : \langle\langle 3\ 4\rangle\ \langle 3\ 6\rangle\rangle \equiv \langle 7\ 9\rangle \tag{1.3}$$

In a value-oriented programming language the computation would be expressed as:

$$\&+ : distl : \langle 3\ \langle 4\ 6\rangle\rangle, \tag{1.4}$$

which means to apply 'distl' to the input and then to apply '+' to every element of the result. In FP we write (1.4) as:

$$\&+ \ @\ distl : \langle 3\ \langle 4\ 6\rangle\rangle. \tag{1.5}$$

The functional expression of (1.5) replaces the two step value expression of (1.4).

Often, functional expressions are built from the inside out, as in LISP. In the next example we derive a function that scales then shifts a vector, i.e., for scalars a, b and a vector $\vec{v}$, compute $a + b\vec{v}$. This FP function will have three arguments, namely a, b and $\vec{v}$. Of course, FP does not use formal parameter names, so they will be designated by the function symbols 1, 2, 3. The first code segment scales $\vec{v}$ by b (defintions are delimited with curly

braces '{}'):

{scaleVec & @ distl @ [2,3]}* (1.6)

The code segment in (1.5) shifts the vector. The completed function is:

{changeVec &+ @ distl @ [1 , scaleVec]} (1.7)

In the derivation of the program we wrote from right to left, first doing the *distl*'s and then composing with the *apply-to-all* functional form. Using an imperative language, such as Pascal, we would write the program from the outside in, writing the loop before inserting the arithmetic operators.

Although FP encourages a recursive programming style, it provides combining forms to avoid explicit recursion. For example, the right insert combining form (!) can be used to write a function that adds up a list of numbers:

!+ : <1 2 3> ≡ 6 (1.8)

The equivalent, recursive function is much longer:

{addNumbers (null -> %0 ; + @ [1, addNumbers @ tl])} (1.9)

The generality of the combining forms encourages hierarchical program development. Unlike APL, which restricts the use of combining forms to certain builtin functions, FP allows combining forms to take any functional expression as an argument.

2. System Description

2.1. Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where the $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax, just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, i.e., 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers (of the type found in FRANZ LISP [Fod80]), quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, **T** and **F**, that correspond to the logical values 'true' and 'false', and the undefined atom **?**, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, <> is also an atom. The following are examples of valid FP objects:

?	1.47	3888888888888
ab	*"CD"*	<1,<2,3>>
<>	**T**	<*a*,<>>

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, e.g., $<1,?> \equiv ?$. This property is the so-called "bottom preserving property" [Ba78].

2.2. Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma{:}x$ is an application and its meaning is the object that results from applying σ to x (i.e., evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand*. The following are examples of applications:

+:<7,8>	≡	15	tl:<1,2,3>	≡	<2,3>
1 :<*a*,*b*,*c*,*d*>	≡	*a*	2 :<*a*,*b*,*c*,*d*>	≡	*b*

2.3. Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\sigma{:}? \equiv ?, \ \forall \ \sigma \in \mathbf{F} \tag{2.1}$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [Mc60]:

$$p_1 \rightarrow e_1 ; \cdots ; p_n \rightarrow e_n ; e_{n+1} \tag{2.2}$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true , . . . , e_n if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that $x, x_i, y, y_i, z_i \in \Omega$.

2.3.1. Structural

Selector Functions

For a nonzero integer μ,

$\mu : x \equiv$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge 0 < \mu \leq k \rightarrow x_\mu;$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge -k \leq \mu < 0 \rightarrow x_{k+\mu+1}; ?$

pick : $\langle n, x \rangle \equiv$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge 0 < n \leq k \rightarrow x_n;$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge -k \leq n < 0 \rightarrow x_{k+n+1}; ?$

The user should note that the function symbols **1,2,3,...** are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$

$x = \langle \rangle \rightarrow \langle \rangle ;$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 1 \rightarrow x_k; ?$

first : $x \equiv$

$x = \langle \rangle \rightarrow \langle \rangle ;$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 1 \rightarrow x_1; ?$

Tail Functions

tl : $x \equiv$

$x = \langle x_1 \rangle \rightarrow \langle \rangle ;$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 2 \rightarrow \langle x_2, \ldots, x_k \rangle ; ?$

tlr : $x \equiv$

$x = \langle x_1 \rangle \rightarrow \langle \rangle ;$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 2 \rightarrow \langle x_1, \ldots, x_{k-1} \rangle ; ?$

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$

$x = <y, <>> \rightarrow <>$;

$x = <y, <z_1, z_2, \ldots, z_k>> \rightarrow <<y, z_1>, \ldots, <y, z_k>>$; *?*

distr : $x \equiv$

$x = <<>, y> \rightarrow <>$;

$x = <<y_1, y_2, \ldots, y_k>, z> \rightarrow <<y_1, z>, \ldots, <y_k, z>>$; *?*

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to **id**. Like **id** it returns its argument as the result, unlike **id** it prints its result on *stdout* – It is the only function with a side effect. *Out* is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

$x = <y, <>> \rightarrow <y>$;

$x = <y, <z_1, z_2, \ldots, z_k>> \rightarrow <y, z_1, z_2, \ldots, z_k>$; *?*

apndr : $x \equiv$

$x = <<>, z> \rightarrow <z>$;

$x = <<y_1, y_2, \ldots, y_k>, z> \rightarrow <y_1, y_2, \ldots, y_k, z>$; *?*

Transpose

trans : $x \equiv$

$x = <<>, \ldots, <>> \rightarrow <>$;

$x = <x_1, x_2, \ldots, x_k> \rightarrow <y_1, \ldots, y_m>$; *?*

where $x_i = <x_{i1}, \ldots, x_{im}> \wedge y_j = <x_{1j}, \ldots, x_{kj}>$,

$1 \leq i \leq k$, $1 \leq j \leq m$.

reverse : $x \equiv$

$x = <> \rightarrow$;

$x = <x_1, x_2, \ldots, x_k> \rightarrow <x_k, \ldots, x_1>$; *?*

Rotate Left and Right

rotl : $x \equiv$

$x=<> \rightarrow <>$; $x=<x_1> \rightarrow <x_1>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 2 \rightarrow <x_2, \ldots, x_k, x_1>$; *?*

rotr : $x \equiv$

$x=<> \rightarrow <>$; $x=<x_1> \rightarrow <x_1>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 2 \rightarrow <x_k, x_1, \ldots, x_{k-2}, x_{k-1}>$; *?*

concat : $x \equiv$

$x=<<x_{11}, \ldots, x_{1k}>, <x_{21}, \ldots, x_{2n}>, \ldots, <x_{m1}, \ldots, x_{mp}>> \wedge k, m, n, p > 0 \rightarrow$
$<x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}>$;*?*

Concatenate removes all occurrences of the null sequence:

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5> (2.3)

pair : $x \equiv$

$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k$ *is even* $\rightarrow <<x_1, x_2>, \ldots, <x_{k-1}, x_k>>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k$ *is odd* $\rightarrow <<x_1, x_2>, \ldots, <x_k>>$; *?*

split : $x \equiv$

$x=<x_1> \rightarrow <<x_1>, <>>$;

$x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow <<x_1, \ldots, x_{\lceil k/2 \rceil}>, <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>$;*?*

iota : $x \equiv$

$x=0 \rightarrow <>$;

$x \in \mathbf{N}^+ \rightarrow <1,2,\ldots,x>$; *?*

2.3.2. Predicate (Test) Functions

atom : $x \equiv x \in$ *atoms* $\rightarrow$ **T**; $x \neq ? \rightarrow$ **F**; *?*

eq : $x \equiv x = <y,z> \wedge y = z \rightarrow$ **T**; $x = <y,z> \wedge y \neq z \rightarrow$ **F**; *?*

Also less than (<), greater than (>), greater than or equal (>=), less than or equal (<=), not equal (~=); '=' is a synonym for **eq**.

null : $x \equiv x = <> \rightarrow$ **T**; $x \neq ? \rightarrow$ **F**; *?*

length : $x \equiv x = \langle x_1, x_2, \ldots, x_k \rangle \rightarrow k; x = \langle\rangle \rightarrow 0; ?$

2.3.3. Arithmetic/Logical

$+ : x \equiv x = \langle y,z \rangle \wedge y,z \text{ are numbers} \rightarrow y+z; ?$
$- : x \equiv x = \langle y,z \rangle \wedge y,z \text{ are numbers} \rightarrow y-z; ?$
$* : x \equiv x = \langle y,z \rangle \wedge y,z \text{ are numbers} \rightarrow y \times z; ?$ $/ : x \equiv x = \langle y,z \rangle \wedge y,z \text{ are numbers} \wedge z \neq 0 \rightarrow y \div z; ?$

And, or, not, xor

and : $\langle x,y \rangle \equiv x = \mathbf{T} \rightarrow y; x = \mathbf{F} \rightarrow \mathbf{F}; ?$

or : $\langle x,y \rangle \equiv x = \mathbf{F} \rightarrow y; x = \mathbf{T} \rightarrow \mathbf{T}; ?$

xor : $\langle x,y \rangle \equiv$
$x = \mathbf{T} \wedge y = \mathbf{T} \rightarrow \mathbf{F}; x = \mathbf{F} \wedge y = \mathbf{F} \rightarrow \mathbf{F};$
$x = \mathbf{T} \wedge y = \mathbf{F} \rightarrow \mathbf{T}; x = \mathbf{F} \wedge y = \mathbf{T} \rightarrow \mathbf{T}; ?$

not : $x \equiv x = \mathbf{T} \rightarrow F; x = \mathbf{F} \rightarrow \mathbf{T}; ?$

2.3.4. Library Routines

sin : $x \equiv x$ is a number $\rightarrow sin(x); ?$

asin : $x \equiv x$ is a number $\wedge \; |x| \leq 1 \rightarrow \sin^{-1}(x); ?$

cos : $x \equiv x$ is a number $\rightarrow cos(x); ?$

acos : $x \equiv x$ is a number $\wedge \; |x| \leq 1 \rightarrow \cos^{-1}(x); ?$

exp : $x \equiv x$ is a number $\rightarrow e^x; ?$

log : $x \equiv x$ is a positive number $\rightarrow ln(x); ?$

mod : $\langle x,y \rangle \equiv x$ **and** y are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor ; ?$

2.4. Functional Forms

Functional forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition*. For two functions ϕ and ψ the form ϕ @ ψ denotes their composition $\phi \circ \psi$:

$$(\phi \; @ \; \psi) : x \equiv \phi:(\psi:x), \quad \forall \quad x \in \Omega \tag{2.4}$$

The *constant* function takes an object parameter:

$$\%x{:}y \equiv y{=}? \rightarrow ?;\ x, \quad \forall \quad x,y \in \Omega \tag{2.5}$$

The function %? always returns ?.

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$(\sigma \ @ \ \tau){:}x \equiv \sigma{:}(\tau{:}x)$

Construction

$[\sigma_1, \ldots, \sigma_n]{:}x \equiv \langle \sigma_1{:}x, \ldots, \sigma_n{:}x \rangle$

Note that construction is also bottom-preserving, e.g.,

$$[+,/]{:}\langle 3,0\rangle = \langle 3,?\rangle = ? \tag{2.6}$$

Condition

$(\xi \text{ -> } \sigma;\ \tau){:}x \equiv$

$(\xi{:}x){=}\mathbf{T} \rightarrow \sigma{:}x;$

$(\xi{:}x){=}\mathbf{F} \rightarrow \tau{:}x;\ ?$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *functional form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus' FP, the conditional form *must* be enclosed in parenthesis, e.g.,

$$\text{(isNegative -> - @ [\%0,id] ; id)} \tag{2.7}$$

Constant

$\%x{:}y \equiv y{=}? \rightarrow ?;\ x, \qquad \forall \quad x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\sigma{:}x \equiv$

$x{=}\langle\rangle \rightarrow e_f{:}x;$

$x{=}\langle x_1\rangle \rightarrow x_1;$

$x{=}\langle x_1, x_2, \ldots, x_k\rangle \wedge k{>}2 \rightarrow \sigma{:}\langle x_1, !\sigma{:}\langle x_2, \ldots, x_k\rangle\rangle;\ ?$

e.g., $!+{:}\langle 4,5,6\rangle{=}15.$

If σ has a right identity element e_f, then $!\sigma:<> = e_f$, e.g.,

$$!+:<>=0 \text{ and } !*:<>=1 \tag{2.8}$$

Currently, identity functions are defined for + (0), − (0), * (1), / (1), also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom (?).

Tree Insert

$| \sigma : x \equiv$

$x=<> \rightarrow e_f:x;$

$x=<x_1> \rightarrow x_1;$

$x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow$

$\sigma : <| \sigma : <x_1, \ldots, x_{\lceil k/2 \rceil}> , | \sigma : <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>;?$

e.g.,

$$|+:<4,5,6,7> \equiv +:<+:<4,5>,+:<6,7>> \equiv 15 \tag{2.9}$$

Tree insert uses the same identity functions as right insert.

Apply to All

$\&\sigma: x \equiv$

$x=<> \rightarrow <>;$

$x=<x_1, x_2, \ldots, x_k> \rightarrow <\sigma:x_1, \ldots, \sigma:x_k>; ?$

While

(**while** ξ σ):$x \equiv$

$\xi:x=\mathbf{T} \rightarrow$ (**while** ξ σ):(σ:x);

$\xi:x=\mathbf{F} \rightarrow x; ?$

2.5. User Defined Functions

An FP definition is entered as follows:

$$\{\textit{fn-name fn-form}\}, \tag{2.10}$$

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid functional form, including a single primitive or defined function. For example the function

$$\{\textit{factorial } !* @ \textit{ iota}\} \tag{2.11}$$

is the non-recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x, \quad \forall \ x \in \Omega$.

References to undefined functions bottom out:

$$f:x \equiv ?\forall\ x \in \Omega, f \notin F \tag{2.12}$$

3. Getting on and off the System

Startup FP from the shell by entering the command:

/usr/local/fp.

The system will prompt you for input by indenting over six character positions. Exit from FP (back to the shell) with a control/D (^D).

3.1. Comments

A user may end any line (including a command) with a comment; the comment character is '#'. The interpreter will ignore any character after the '#' until it encounters a newline character or end-of-file, whichever comes first.

3.2. Breaks

Breaks interrupt any work in progress causing the system to do a FRANZ reset before returning control back to the user.

3.3. Non-Termination

LISP's namestack may, on occasion, overflow. FP responds by printing "non-terminating" and returning bottom as the result of the application. It does a FRANZ reset before returning control to the user.

4. System Commands

System commands start with a right parenthesis and they are followed by the command-name and possibly one or more arguments. All this information *must be typed on a single line*, and any number of spaces or tabs may be used to separate the components.

4.1. Load

Redirect the standard input to the file named by the command's argument. If the file doesn't exist then FP appends '.fp' to the file-name and retries the open (error if the file doesn't exist). This command allows the user to read in FP function definitions from a file. The user can also read in applications, but such operation is of little utility since none of the input is echoed at the terminal. Normally, FP returns control to the user on an end-of-file. It will also do so whenever it does a FRANZ reset, e.g., whenever the user issues a break, or whenever the system encounters a non-terminating application.

4.2. Save

Output the source text for all user-defined functions to the file named by the argument.

4.3. Csave and Fsave

These commands output the lisp code for all the user-defined functions, including the original source-code, to the file named by the argument. Csave pretty prints the code, Fsave does not. Unless the user wishes to examine the code, he should use 'fsave'; it is about ten times faster than 'csave', and the resulting file will be about three times smaller.

These commands are intended to be used with the liszt compiler and the 'cload' command, as explained below.

4.4. Cload

This command loads or fasls in the file shown by the argument. First, FP appends a '.o' to the file-name, and attempts a load. Failing that, it tries to load the file named by the argument. If the user outputs his function definitions using fsave or csave, and then compiles them using liszt, then he may fasl in the compiled code and speed up the execution of his defined functions by a factor of 5 to 10.

4.5. Pfn

Print the source text(s) (at the terminal) for the user-defined function(s) named by the argument(s) (error if the function doesn't exist).

4.6. Delete

Delete the user-defined function(s) named by the argument (error if the function doesn't exist).

4.7. Fns

List the names of all user-defined functions in alphabetical order. Traced functions are labeled by a trailing '@' (see § 4.7 for sample output).

4.8. Stats

The "stats" command has several options that help the user manage the collection of dynamic statistics for functions[1] and functional forms. Option names follow the keyword "stats", e.g., ")stats reset".

The statistic package records the frequency of usage for each function and functional form; also the size[2] of all the arguments for all functions and functional expressions. These two measures allow the user to derive the average argument size per call. For functional forms the package tallies the frequency of each functional argument. Construction has an additional statistic that tells the number of functional arguments involved in the construction.

Statistics are gathered whenever the mode is on, except for applications that "bottom out" (i.e., return bottom – ?). Statistic collection slows the system down by ×2 to ×4. The following printout illustrates the use of the statistic package (user input is emboldened):

---

[1] Measurement of user-defined functions is done with the aid of the trace package, discussed in § 4.9.

[2] "Size" is the top-level length of the argument, for most functions. Exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

```
)stats on

Stats collection turned on.

+:<3 4>
7
!* @ iota :3
6
)stats print

plus:     times    1

times:    times    2

iota:     times    1

insert:   times    1              size 3

                Functional Args
            Name                        Times
            times                  1

compos:   times    1              size 1

                Functional Args
            Name                        Times
            insert                 1
            iota                           1
```

4.8.1. On

Enable statistics collection.

4.8.2. Off

Disable statistics collection. The user may selectively collect statistics using the on and off commands.

4.8.3. Print

Print the dynamic statistics at the terminal, or, output them to a file. The latter option requires an additional argument, e.g., ")stats print fooBar" prints the stats to the file "fooBar".

4.8.4. Reset

Reset the dynamic statistics counters. To prevent accidental loss of collected statistics, the system will query the user if he tries to reset the counters without first outputting the data

(the system will also query the user if he tries to log out without outputting the data).

4.9. Trace

Enable or disable the tracing and the dynamic measurement of the user defined functions named by the argument(s). The first argument tells whether to turn tracing off or on and the others give the name of the functions affected. The tracing and untracing commands are independent of the dynamic statistics commands. This command is cumulative e.g., ')trace on f1', followed by ')trace on f2' is equivalent to ')trace on f1 f2'.

FP tracer output is similar to the FRANZ tracer output: function entries and exits, call level, the functional argument (remember that FP functions have only one argument!), and the result, are printed at the terminal:

```
                )pfn fact

{fact (eq0 -> %1 ; * @ [id, fact @ s1])}
                )fns

eq0             fact s1

                )trace on fact
                )fns

eq0             fact@     s1

                fact : 2

1 >Enter> fact [2]
|2 >Enter> fact [1]
| 3 >Enter> fact [0]
| 3 <EXIT< fact 1
|2 <EXIT< fact 1
1 <EXIT< fact 2

2
```

4.10. Timer

FP provides a simple timing facility to time top-level applications. The command ")timer on" puts the system in timing mode, ")timer off" turns the mode off (the mode is initially off). While in timing mode, the system reports CPU time, garbage collection time, and elapsed time, in seconds. The timing output follows the printout of the result of the application.

4.11. Script

Open or close a script file. The first argument gives the option, the second the optional script file-name. The "open" option causes a new script-file to be opened and any currently open script file to be closed. If the file cannot be opened, FP sends and error message and, if a script file was already opened, it remains open. The command ")script close" closes an

open script file. The user may elect to append script output to the script-file with the append mode.

4.12. Help

Print a short summary of all the system commands:

```
    )help
Commands are:

load <file>                     Redirect input from <file>
save <file>                     Save defined fns in <file>
pfn <fn1> ...                   Print source text of <fn1> ...
delete <fn1> ...                Delete <fn1> ...
fns                             List all functions
stats on/off/reset/print [file] Collect and print dynamic stats
trace on/off <fn1> ...          Start/Stop exec trace of <fn1> ...
timer on/of                     Turn timer on/off
script open/close/append        Open or close a script-file
lisp                            Exit to the lisp system (return with '^D')
debug on/off                    Turn debugger output on/off
csave <file>                    Output Lisp code for all user-defined fns
cload <file>                    Load Lisp code from a file (may be compiled)
fsave <file>                    Same as csave except without pretty-printing
```

4.13. Special System Functions

There are two system functions that are not generally meant to be used by average users.

4.13.1. Lisp

This exits to the lisp system. Use "^D" to return to FP.

4.13.2. Debug

Turns the 'debug' flag on or off. The command ")debug on" turns the flag on, ")debug off" turns the flag off. The main purpose of the command is to print out the parse tree.

5. Programming Examples

We will start off by developing a larger FP program, *mergeSort*. We measure *mergeSort* using the trace package, and then we comment on the measurements. Following *mergeSort* we show an actual session at the terminal.

5.1. MergeSort

The source code for *mergeSort* is:

```
# Use a divide and conquer strategy
{mergeSort | merge}

{merge atEnd @ mergeHelp @ [[], fixLists]}

# Must convert atomic arguments into sequences
# Atomic arguments occur at the leaves of the execution tree
{fixLists &(atom -> [id] ; id)}

# Merge until one or both input lists are empty
{mergeHelp (while and @ &(not@null) @ 2
                  (firstIsSmaller -> takeFirst ;
                                     takeSecond))}

# Find the list with the smaller first element
{firstIsSmaller < @ [1@1@2, 1@2@2]}

# Take the first element of the first list
{takeFirst [apndr@[1,1@1@2], [tl@1@2, 2@2]]}

# Take the first element of the second list
{takeSecond [apndr@[1,1@2@2], [1@2, tl@2@2]]}

# If one list isn't null, then append it to the
# end of the merged list
{atEnd (firstIsNull -> concat@[1,2@2] ;
                       concat@[1,1@2])}

{firstIsNull null@1@2}
```

The merge sort algorithm uses a divide and conquer strategy; it splits the input in half, recursively sorts each half, and then merges the sorted lists. Of course, all these sub-sorts can execute in parallel, and the tree-insert (|) functional form expresses this concurrency. *Merge* removes successively larger elements from the heads of the two lists (either *takeFirst* or *takeSecond*) and appends these elements to the end of the merged sequence. *Merge* terminates when one sequence is empty, and then *atEnd* appends any remaining non-empty sequence to the end of the merged one.

On the next page we give the trace of the function *merge*, which information we can use to determine the structure of *merge*'s execution tree. Since the tree is well-balanced, many of the *merge*'s could be executed in parallel. Using this trace we can also calculate the average length of the arguments passed to *merge*, or a distribution of argument lengths. This information is useful for determining communication costs.

```
        )trace on merge

        mergeSort : <0 3 -2 1 11 8 -22 -33>
| 3 >Enter> merge [<0 3>]
| 3 <EXIT< merge <0 3>
| 3 >Enter> merge [<-2 1>]
| 3 <EXIT< merge <-2 1>
|2 >Enter> merge [<<0 3> <-2 1>>]
|2 <EXIT< merge <-2 0 1 3>
| 3 >Enter> merge [<11 8>]
| 3 <EXIT< merge <8 11>
| 3 >Enter> merge [<-22 -33>]
| 3 <EXIT< merge <-33 -22>
|2 >Enter> merge [<<8 11> <-33 -22>>]
|2 <EXIT< merge <-33 -22 8 11>
1 >Enter> merge [<<-2 0 1 3> <-33 -22 8 11>>]
1 <EXIT< merge <-33 -22 -2 0 1 3 8 11>

<-33 -22 -2 0 1 3 8 11>
```

5.2. FP Session

User input is **emboldened**, terminal output in Roman script.

fp

```
FP, v. 4.1 11/31/82
        )load ex_man
{all_le}
{sort}
{abs_val}
{find}
{ip}
{mm}
{eq0}
{fact}
{sub1}
{alt_fnd}
{alt_fact}
        )fns

abs_val   all_le   alt_fact   alt_fnd   eq0   fact   find
ip        mm       sort       sub1

        abs_val : 3

3

        abs_val : -3

3

        abs_val : 0

0

        abs_val : <-5 0 66>

?

        &abs_val : <-5 0 66>

<5 0 66>

        )pfn abs_val

{abs_val ((> @ [id,%0]) -> id ; (- @ [%0,id]))}

        [id,%0] : -3

<-3 0>
```

```
        [%0,id] : -3
<0 -3>
        - @ [%0,id] : -3
3
        all_le : <1 3 5 7>
T
        all_le : <1 0 5 7>
F
        )pfn all_le
{all_le ! and @ &<= @ distl @ [1,tl]}
        distl @ [1,tl] : <1 2 3 4>
<<1 2> <1 3> <1 4>>
        &<= @ distl @ [1,tl] : <1 2 3 4>
<T T T>
        ! and : <F T T>
F
        ! and : <T T T>
T
        sort : <3 1 2 4>
<1 2 3 4>
        sort : <1>
<1>
        sort : <>
?
        sort : 4
?
        )pfn sort
```

```
{sort (null @ tl -> [1] ; (all_le -> apndl @ [1,sort@tl]; sort@rotl))}
        fact : 3
6
        )pfn fact sub1 eq0
{fact (eq0 -> %1 ; *@[id , fact@sub1])}
{sub1 -@[id,%1]}
{eq0 = @ [id,%0]}
        &fact : <1 2 3 4 5>
<1 2 6 24 120>
        eq0 : 3
F
        eq0 : <>
F
        eq0 : 0
T
        sub1 : 3
2
        %1 : 3
1
        alt_fact : 3
6
        )pfn alt_fact
{alt_fact !* @ iota}
        iota : 3
<1 2 3>
        !* @ iota : 3
6
```

```
        !+ : <1 2 3>
6
        find : <3 <3 4 5>>
T
        find : <<> <3 4 <>>>
T
        find : <3 <4 5>>
F
        )pfn find
(find (null@2 -> %F ; (=@[1,1@2] -> %T ; find@[1,tl@2]))}
        [1,tl@2] : <3 <3 4 5>>
<3 <4 5>>
        [1,1@2] : <3 <3 4 5>>
<3 3>
        alt_fnd : <3 <3 4 5>>
T
        )pfn alt_fnd
(alt_fnd ! or @ &eq @ distl }
        distl : <3 <3 4 5>>
<<3 3> <3 4> <3 5>>
        &eq @ distl : <3 <3 4 5>>
<T F F>
        !or : <T F T>
T
        !or : <F F F>
F
        )delete alt_fnd
```

```
        )fns

abs_val   all_le   alt_fact   eq0   fact   find   ip
mm        sort     sub1

        alt_fnd : <3 <3 4 5>>

alt_fnd not defined

?
        {g g}

{g}
        g : 3

non-terminating

?

[Return to top level]

FP, v. 4.0 10/8/82
        [+,*] : <3 4>

<7 12>

                [+,* : <3 4>

         syntax error:

         [+,* : <3 4>
                ^

        ip : <<3 4 5> <5 6 7>>

74

        )pfn ip

{ip !+ @ &* @ trans}

        trans : <<3 4 5> <5 6 7>>

<<3 5> <4 6> <5 7>>

        &* @ trans : <<3 4 5> <5 6 7>>

<15 24 35>

        mm : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<3 4> <5 6>>
```

```
      )pfn mm
{mm &&ip @ &distl @ distr @[1,trans@2]}
      [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
<<<1 0> <0 1>> <<3 4> <5 6>>>
      distr : <<<1 0> <0 1>> <<3 4> <5 6>>>
<<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>
      &distl : <<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>
<<<<1 0> <3 4>> <<1 0> <5 6>>> <<<0 1> <3 4>> <<0 1> <5 6>>>>
            &ip @ &dist & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
      syntax error:
      [+,* : <3 4>
      &ip @ &distl & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
      &ip @ &distl @ distr @ [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
?
```

6. Implementation Notes

FP was written in 3000 lines of FRANZ LISP [Fod 80]. Table 1 breaks down the distribution of the code by functionality.

Functionality	% (bytes)
compiler	34
user interface	32
dynamic stats	16
primitives	14
miscellaneous	3

Table 1

6.1. The Top Level

The top-level function *runFp* starts up the subsystem by calling the routine *fpMain*, that takes three arguments:

(1) A boolean argument that says whether debugging output will be enabled.

(2) A Font identifier. Currently the only one is supported 'asc (ASCII).

(3) A boolean argument that identifies whether the interpreter was invoked from the shell. If so then all exits from FP return the user back to the shell.

The compiler converts the FP functions into LISP equivalents in two stages: first it forms the parse tree, and then it does the code generation.

6.2. The Scanner

The scanner consists of a main routine, *get_tkn*, and a set of action functions. There exists one set of action functions for each character font (currently only ASCII is supported). All the action functions are named *scan$<font>*, where *<font>* is the specified font, and each is keyed on a particular character (or sometimes a particular character-type – e.g., a letter or a number). *get_tkn* returns the token type, and any ancillary information, e.g., for the token "name" the name itself will also be provided. (See Appendix C for the font-token name correspondences). When a character has been read the scanner finds the action function by doing a

(get 'scan$ <font> <char>)

A syntax error message will be generated if no action exists for the particular character read.

6.3. The Parser

The main parsing function, *parse*, accepts a single argument, that identifies the parsing context, or type of construct being handled. Table 2 shows the valid parsing contexts.

id	construct
top_lev	initial call
constr$$	construction
compos$$	composition
alpha$$	apply-to-all
insert$$	insert
ti$$	tree insert
arrow$$	affirmative clause of conditional
semi$$	negative clause of conditional
lparen$$	parenthetic expr.
while$$	while

Table 2, Valid Parsing Contexts

For each type of token there exists a set of parse action functions, of the name *p$<tkn-name>*. Each parse-action function is keyed on a valid context, and it is looked up in the same manner as scan action functions are looked up. If an action function cannot be found, then there is a syntax error in the source code. Parsing proceeds as follows: initially *parse* is called from the top-level, with the context argument set to *"top_lev"*. Certain tokens cause parse to be recursively invoked using that token as a context. The result is the parse tree.

6.4. The Code Generator

The system compiles FP source into LISP source. Normally, this code is interpreted by the FRANZ LISP system. To speed up the implementation, there is an option to compile into machine code using the *liszt* compiler [Joy 79]. This feature improves performance tenfold, for some programs.

The compiler expands all functional forms into their LISP equivalents instead of inserting calls to functions that generate the code at run-time. Otherwise, *liszt* would be ineffective in speeding up execution since all the functional forms would be executed interpretively. Although the amount of code generated by an expanding compiler is 3 or 4 times greater than would be generated by a non-expanding compiler, even in interpreted mode the code runs twice as quickly as unexpanded code. With *liszt* compilation this performance advantage increases to more than tenfold.

A parse tree is either an atom or a hunk of parse trees. An atomic parse-tree identifies either an fp built-in function or a user defined function. The hunk-type parse tree represents functional forms, e.g., compose or construct. The first element identifies the functional form and the other elements are its functional parameters (they may in turn be functional forms). Table 3 shows the parse-tree formats.

Form	Format
user-defined	<atom>
fp builtin	<atom>
apply-to-all	{*alpha$$* Φ}
insert	{*insert$$* Φ}
tree insert	{*ti$$* Φ}
select	{*select$$* μ}
constant	{*constant$$* μ}
conditional	{*condit$$* Φ_1 Φ_2 Φ_3}
while	{*while$$* Φ_1 Φ_2}
compose	{*compos$$* Φ_1 Φ_2}
construct	{*constr$$* Φ_1 Φ_2 , . . . , Φ_n *nil*}

Note: Φ and the Φ_k are parse-trees and μ is an optionally signed integer constant.

Table 3, Parse-Tree Formats

6.5. Function Definition and Application

Once the code has been generated, then the system defines the function via *putd*. The source code is placed onto a property list, *'sources*, to permit later access by the system commands.

For an application, the indicated function is compiled and then defined, only temporarily, as *tmp$$*. The single argument is read and *tmp$$* is applied to it.

6.6. Function Naming Conventions

When the parser detects a named primitive function, it returns the name *<name>$fp*, where *<name>* is the name that was parsed (all primitive function-names end in *$fp*). See Appendix D for the symbolic (e.g., compose, +) function names.

Any name that isn't found in the list of builtin functions is assumed to represent a user-defined function; hence, it isn't possible to redefine FP primitive functions. FP protects itself from accidental or malicious internal destruction by appending the suffix "*_fp*" to all user-defined function names, before they are defined.

6.7. Measurement Impelementation

This work was done by Dorab Patel at UCLA. Most of the measurement code is in the file 'fpMeasures.l'. Many of the remaining changes were effected in 'primFp.l', to add calls on the measurement package at run-time; to 'codeGen.l', to add tracing of user defined functions; to 'utils.l', to add the new system commands; and to 'fpMain.l', to protect the user from forgetting to output statistics when he leaves FP.

6.7.1. Data Structures

All the statistics are in the property list of the global symbol *Measures*. Associated with each each function (primitive or user-defined, or functional form) is an indicator; the statistics gathered for each function are the corresponding values. The names corresponding to primitive functions and functional forms end in '$fp' and the names corresponding to user-defined functions end in '_fp'. Each of the property values is an association list:

```
(get 'Measures 'rotl$fp) ==> ((times . 0) (size . 0))
```

The car of the pair is the name of the statistic (i.e., times, size) and the cdr is the value. There is one exception. Functional forms have a statistic called funargtyp. Instead of being a dotted pair, it is a list of two elements:

```
(get 'Measures 'compose$fp) ==>
((times . 2) (size . 4) (funargtyp ((select$fp . 2) (sub$fp . 2))))
```

The car is the atom 'funargtyp' and the cdr is an alist. Each element of the alist consists of a functional argument-frequency dotted pair.

The statistic packages uses two other global symbols. The symbol DynTraceFlg is non-nil if dynamic statistics are being collected and is nil otherwise. The symbol TracedFns is a list (initially nil) of the names of the user functions being traced.

6.7.2. Interpretation of Data Structures

6.7.2.1. Times

The number of times this function has been called. All functions and functional forms have this statistic.

6.7.2.2. Size

The sum of the sizes of the arguments passed to this function. This could be divided by the times statistic to give the average size of argument this function was passed. With few exceptions, the size of an object is its top-level length (note: version 4.0 defined the size as the total number of *atoms* in the object); the empty sequence, "<>", has a size of 0 and all other atoms have size of one. The exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

This statistic is not collected for some primitive functions (mainly binary operators like +,-,*).

6.7.2.3. Funargno

The number of functional arguments supplied to a functional form.

Currently this statistic is gatherered only for the construction functional form.

6.7.2.4. Funargtyp

How many times the named function was used as a functional parameter to the particular functional form.

6.8. Trace Information

The level number of a call shows the number of steps required to execute the function on an ideal machine (i.e., one with unbounded resources). The level number is calculated under an assumption of infinite resources, and the system evaluates the condition of a conditional before evaluating either of its clauses. The number of functions executed at each level can give an idea of the distribution of parallelism in the given FP program.

7. Acknowledgements

Steve Muchnick proposed the initial construction of this system. Bob Ballance added some of is own insights, and John Foderaro provided helpful hints regarding effective use of the FRANZ LISP system [Fod80]. Dorab Patel [Pat81] wrote the dynamic trace and statistics package and made general improvements to the user interface. Portions of this manual were excerpted from the *COMPCON-83 Digest of Papers*[3].

8. References

[Bac78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, Turing Award Lecture, 21, 8 (August 1978), 613-641.

[Fod80]
John K. Foderaro, "The FRANZ LISP Manual," University of California, Berkeley, California, 1980.

[Joy79]
W.N. Joy, O. Babaoglu, "UNIX Programmer's Manual," November 7, 1979, Computer Science Division, University of California, Berkeley, California.

[Mc60]
J. McCarthy, "Recursive Functions of Symbolic expressions and their Computation by Machine," Part I, *CACM* 3, 4 (April 1960), 184-195.

[Pat80]
Dorab Ratan Patel, "A System Organization for Applicative Programming," M.S Thesis, University of California, Los Angeles, California, 1980.

[Pat81]
Dorab Patel, "Functional Language Interpreter User Manual," University of California, Los Angeles, California, 1981.

---

[3] Scott B. Baden and Dorab R. Patel, "Berkeley FP – Experiences With a Functional Programming Language",

Appendix A: Local Modifications

1. Character Set Changes

Backus [Ba78] used some characters that do not appear on our ASCII terminals, so we have made the following substitutions:

constant	$\bar{x}$	%x
insert	/	!
apply-to-all	α	&
composition	$\circ$	@
arrow	$\rightarrow$	->
empty set	ϕ	<>
bottom	$\perp$	?
divide	$\div$	/
multiply	$\times$	*

2. Syntactic Modifications

2.1. While and Conditional

While and conditional functional expressions *must* be enclosed in parenthesis, e.g.,

(**while** *f g*)

(*p* -> *f*; *g*)

2.2. Function Definitions

Function definitions are enclosed by curly braces; they consist of a name-definition pair, separated by blanks. For example:

{**fact** !* @ iota}

defines the function **fact** (the reader should recognize this as the non-recursive factorial function).

2.3. Sequence Construction

It is not necessary to separate elements of a sequences with a comma; a blank will suffice:

<1,2,3> ≡ <1 2 3>

For nested sequences, the terminating right angle bracket acts as the delimiter:

<<1,2,3>,<4,5,6>> ≡ <<1 2 3><4 5 6>>

3. User Interface

We have provided a rich set of commands that allow the user to catalog, print, and delete functions, to load them from a file and to save them away. The user may generate script files, dynamically trace and measure functional expression execution, generate debugging output, and, temporarily exit to the FRANZ LISP system. A command must begin with a right parenthesis. Consult Appendix C for a complete description of the command syntax.

Debugging in FP is difficult; all undefined results map to a single atom – *bottom* ("?"). To pinpoint the cause of an error the user can use the special debugging output function, **out**, or the tracer.

4. Additions and Ommissions

Many relational functions have been added: <, >, =, ≠, ≤, ≥; their syntax is: <, >, =, ~=, <=, >=. Also added are the **iota** function (This is the APL iota function an n-element sequence of natural numbers) and the exclusive OR (⊕) function.

Several new structural functions have been added: **pair** pairs up successive elements of a sequence, **split** splits a sequence into two (roughly) equal halves, **last** returns the last element of the sequence (<> if the sequence is empty), **first** returns the first element of the sequence (<> if it is empty), and **concat** concatenates all subsequences of a sequence, squeezing out null sequences (<>). **Front** is equivalent to **tlr**. **Pick** is a parameterized form of the selector function; the first component of the argument selects a single element from the second component. **Out** is the only side-effect function; it is equivalent to the **id** function but it also prints its argument out at the terminal. This function is intended to be used only for debugging.

One new functional form has been added, tree insert. This functional form breaks up the the argument into two roughly equal pieces applying itself recursively to the two halves. The functional parameter is applied to the result.

The binary-to-unary functions ('**bu**') has been omitted.

Seven mathematical library functions have been added: sin, cos, asin ($\sin^{-1}$), acos ($\cos^{-1}$), log, exp, and mod (the remainder function)

Appendix B: FP Grammar

I. BNF Syntax

fpInput →	(fnDef | application | fpCmd[a])* | '^D'
fnDef →	'{' name funForm '}'
application →	funForm ':' object
name →	letter (letter | digit | '_')*
nameList →	(name)*
object →	atom | fpSequence | '?'
fpSequence →	'<' (ϵ | object ((',' | ' ') object)*) '>'
atom →	'T' | 'F' | '<>' | '"' (ascii-char)* '"' | (letter | digit)* | number
funForm →	simpFn | composition | construction | conditional | constantFn | insertion | alpha | while | '(' funForm ')'
simpFn →	fpDefined | fpBuiltin
fpDefined →	name
fpBuiltin →	selectFn | 'tl' | 'id' | 'atom' | 'not' | 'eq' | relFn | 'null' | 'reverse' | 'distl' | 'distr' | 'length' | binaryFn | 'trans' | 'apndl' | 'apndr' | 'tlr' | 'rotl' | 'rotr' | 'iota' | 'pair' | 'split' | 'concat' | 'last' | 'libFn'
selectFn →	(ϵ | '+' | '-') unsignedInteger
relFn →	'<=' | '<' | '=' | '~=' | '>' | '>='
binaryFn →	'+' | '-' | '*' | '/' | 'or' | 'and' | 'xor'
libFn →	'sin' | 'cos' | 'asin' | 'acos' | 'log' | 'exp' | 'mod'
composition →	funForm '@' funForm
construction →	'[' formList ']'
formList →	ϵ | funForm (',' funForm)*
conditional →	'(' funForm '->' funForm ';' funForm ')'
constantFn →	'%' object
insertion →	'!' funForm | '|' funForm
alpha →	'&' funForm
while →	'(' 'while' funForm funForm ')'

II. Precedences

1.	%, !, &	(highest)
2.	@	
3.	[· · ·]	
4.	-> · · · ; · · ·	
5.	while	(least)

[a] Command Syntax is listed in Appendix C.

Appendix C: Command Syntax

All commands begin with a right parenthesis (")").

)fns
)pfn <nameList>
)load <UNIX file name>
)cload <UNIX file name>
)save <UNIX file name>
)csave <UNIX file name>
)fsave <UNIX file name>
)delete <nameList>
)stats on
)stats off
)stats reset
)stats print [UNIX file name]
)trace on <nameList>
)trace off <nameList>
)timer on
)timer off
)debug on
)debug off
)script open <UNIX file name>
)script close
)script append <UNIX file name>
)help
)lisp

Appendix D: Token-Name Correspondences

Token	Name
[	lbrack$$
]	rbrack$$
{	lbrace$$
}	rbrace$$
(	lparen$$
)	rparen$$
@	compos$$
!	insert$$
|	ti$$
&	alpha$$
;	semi$$
:	colon$$
,	comma$$
+	builtin$$
+ μ[a]	select$$
*	builtin$$
/	builtin$$
=	builtin$$
-	builtin$$
->	arrow$$
- μ	select$$
>	builtin$$
>=	builtin$$
<	builtin$$
<=	builtin$$
~=	builtin$$
%o[b]	constant$$

[a] μ is an optionally signed integer constant.

[b] o is any FP object.

Appendix E: Symbolic Primitive Function Names

The scanner assigns names to the alphabetic primitive functions by appending the string "$fp" to the end of the function name. The following table designates the naming assignments to the non-alphabetic primitive function names.

Function	Name
+	plus$fp
-	minus$fp
*	times$fp
/	div$fp
=	eq$fp
>	gt$fp
>=	ge$fp
<	lt$fp
<=	le$fp
~=	ne$fp

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
structured programming, control flow, programming

ABSTRACT

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

- statement grouping
- **if-else** and **switch** for decision-making
- **while**, **for**, **do**, and **repeat-until** for looping
- **break** and **next** for controlling loop exits

and some "syntactic sugar":

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of >, >=, etc., into .GT., .GE., etc.
- **return**(expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other environments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

This paper is a revised and expanded version of oe published in *Software—Practice and Experience*, October 1975. The Ratfor described here is the one in use on UNIX and GCOS at Bell Laboratories, Murray Hill, N. J.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N−1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran.* Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects wili be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1;
              return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
              call error(5hx>100)
              err = 1
              return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like "x>100"), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of H's: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
      call error("x>100")
      err = 1
      return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
      write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
      { sw = 0; write(6, 1) a, b }
else
      { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
      if (a .gt. b) goto 10
            sw = 0
            write(6, 1) a, b
            goto 20
10    sw = 1
      write(6, 1) b, a
20    ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable f is to be set to −1 if x is less than zero, to +1 if x is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        - - -
else if (...)
        - - -
else if (...)
        - - -
 ...
else
        - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {

        case expr1 :
                statements
        case expr2, expr3 :
                statements
        ...
        default:
                statements
}
```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1, expr2,* and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression,* and there is a **default** section, the statements with it are done; if there is no **default,** nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch.**)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
      do 10 i = 1, n
              x(i) = 0.0
              y(i) = 0.0
              z(i) = 0.0
10    continue
```

The syntax is:

do *legal-Fortran-DO-text*
    Ratfor statement

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do.**

The *Ratfor statement* part will often be enclosed in braces, but as with the **if,** a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with I set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k {
                ---
        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
    # returns sin(x) to accuracy e, by
    # sin(x) = x - x**3/3! + x**5/5! - ...

    sin = x
    term = x

    i = 3
    while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
    }

    return
    end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal Fortran condition)
        Ratfor statement
```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
    ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
100   if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while**. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100;
  i=i+2) {
    term = -term * x**2 / float(i*(i-1))
    sin = sin + term
}
```

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
    Ratfor statement

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
    if (card(i) != blank)
        break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
          I = 81 - J
          IF (CARD(I) .NE. BLANK) GO TO 11
   10 CONTINUE
      I = 0
   11 ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
        Ratfor statement
until (legal Fortran condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value -1.

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1) {
                        equal = 1
                        return
                }
        equal = 0
        return
        end
```

In many languages (e.g., PL/I) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function F, **return**(expression) is equivalent to

```
{ F = expression; return }
```

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1)
                        return(1)
        return(0)
        end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
=  +  -  *  ,  |  &  (  _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	¬	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	{	]	}
$(	{	$)	}

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS      100
define COLS 50
dimension a(ROWS), b(ROWS, COLS)
      if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define  YES     1
define  NO      0
define  EOS     -1
define  ARB             100

# equal _ compare str1 to str2;
#       return YES if equal, NO if not
        integer function equal(str1, str2)
        integer str1(ARB), str2(ARB)
        integer i

        for (i = 1; str1(i) == str2(i);
        i = i + 1)
             if (str1(i) == EOS)
                  return(YES)
        return(NO)
        end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
     include commonblocks
     ...
     end

suroutine y
     include commonblocks
     ...
     end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog  : stat
      | prog  stat
stat  : if (...) stat
      | if (...) stat else stat
      | while (...) stat
      | for (...; ...; ...) stat
      | do ... stat
      | repeat stat
      | repeat stat until (...)
      | switch (...) { case ...: prog ...
                       default: prog }
      | return
      | break
      | next
      | digits  stat
      | { prog }
      | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L     continue
```

is generated, unless there is an **else** clause, in which case the code is

```
      goto L+1
L     continue
```

In this latter case, the code

```
L+1   continue
```

is produced after the *statement* part of the **else**. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
      if (.not. (i .gt. 0)) goto 100
      x = a
100   continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form *c*v±c;* avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read.* The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience,* October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter,* August 1975.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM,* July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys,* December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM,* August 1971.

Appendix: Usage on UNIX and GCOS.

Beware — local customs vary. Check with a native before going into the jungle.

UNIX

The program **ratfor** is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **-6x**, which uses x as a continuation character in column 6 (UNIX uses **&** in column 1), and **-C**, which causes Ratfor comments to be copied into the generated Fortran.

The program **rc** provides an interface to the **ratfor** command which is much the same as **cc**. Thus

rc [options] files

compiles the files specified by **files**. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags **-C** and **-6x** described above are recognized, as are

-c	compile only; don't load
-f	save intermediate Fortran .f files
-r	Ratfor only; implies -c and -f
-2	use big Fortran compiler (for large programs)
-U	flag undeclared variables (not universally available)

Other flags are passed on to the loader.

GCOS

The program **./ratfor** is the bare translator, and is identical to the UNIX version, except that the continuation convention is **&** in column 6. Thus

./ratfor files >output

translates the Ratfor source on **files** and collects the generated Fortran on file 'output' for subsequent processing.

./rc provides much the same services as **rc** (within the limitations of GCOS), regrettably with a somewhat different syntax. Options recognized by **./rc** include

name	Ratfor source or library, depending on type
h=/name	make TSS H* file (runnable version); run as /name
r=/name	update and use random library
a=	compile as ascii (default is bcd)
C=	copy comments into Fortran
f=name	Fortran source file
g=name	gmap source file

Other options are as specified for the **./cc** command described in [4].

TSO, TSS, and other systems

Ratfor exists on various other systems; check with the author for specifics.

The FRANZ LISP Manual

by

John K. Foderaro

Keith L. Sklower

Kevin Layer

June 1983

A document in
four movements

Overture

*A chorus of students under the direction of Richard Fateman have contributed to building FRANZ LISP from a mere melody into a full symphony . The major contributors to the initial system were Mike Curry, John Breedlove and Jeff Levinsky. Bill Rowan added the garbage collector and array package. Tom London worked on an early compiler and helped in overall system design. Keith Sklower has contributed much to FRANZ LISP, adding the bignum package and rewriting most of the code to increase its efficiency and clarity. Kipp Hickman and Charles Koester added hunks. Mitch Marcus added *rset, evalhook and evalframe. Don Cohen and others at Carnegie-Mellon made some improvements to evalframe and provided various features modelled after UCI/CMU PDP-10 Lisp and Interlisp environments (editor, debugger, top-level). John Foderaro wrote the compiler, added a few functions, and wrote much of this manual. Of course, other authors have contributed specific chapters as indicated. Kevin Layer modified the compiler to produce code for the Motorola 68000, and helped make FRANZ LISP pass "Lint".*
This manual may be supplemented or supplanted by local chapters representing alterations, additions and deletions. We at U.C. Berkeley are pleased to learn of generally useful system features, bug fixes, or useful program packages, and we will attempt to redistribute such contributions.

Work reported herein was supported in part by the U. S. Department of Energy, Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358, and the National Science Foundation under Grant No. MCS 7807291

Score

First Movement *(allegro non troppo)*

1. FRANZ LISP
   Introduction to FRANZ LISP, details of data types, and description of notation
2. Data Structure Access
   Functions for the creation, destruction and manipulation of lisp data objects.
3. Arithmetic Functions
   Functions to perform arithmetic operations.
4. Special Functions
   Functions for altering flow of control. Functions for mapping other functions over lists.
5. I/O Functions
   Functions for reading and writing from ports. Functions for the modification of the reader's syntax.
6. System Functions
   Functions for storage management, debugging, and for the reading and setting of global Lisp status variables. Functions for doing UNIX-specific tasks such as process control.

Second Movement *(Largo)*

7. The Reader
   A description of the syntax codes used by the reader. An explanation of character macros.
8. Functions, Fclosures, and Macros
   A description of various types of functional objects. An example of the use of foreign functions.
9. Arrays and Vectors
   A detailed description of the parts of an array and of Maclisp compatible arrays.
10. Exception Handling
    A description of the error handling sequence and of autoloading.

Third Movement *(Scherzo)*

11. The Joseph Lister Trace Package
    A description of a very useful debugging aid.
12. Liszt, the lisp compiler
    A description of the operation of the compiler and hints for making functions compilable.
13. CMU Top Level and File Package
    A description of a top level with a history mechanism and a package which helps you keep track of files of lisp functions.
14 Stepper
    A description of a program which permits you to put breakpoints in lisp code and to single step it. A description of the evalhook and funcallhook mechanism.
15 Fixit
    A program which permits you to examine and modify evaluation stack in order to fix bugs on the fly.
16 Lisp Editor
    A structure editor for interactive modification of lisp code.

Final Movement *(allegro)*

Appendix A - Function Index
Appendix B - List of Special Symbols
Appendix C - Short Subjects
    Garbage collector, Debugging, Default Top Level

CHAPTER 1

FRANZ LISP

1.1. FRANZ LISP[†] was created as a tool to further research in symbolic and algebraic manipulation, artificial intelligence, and programming languages at the University of California at Berkeley. Its roots are in a PDP-11 Lisp system which originally came from Harvard. As it grew it adopted features of Maclisp and Lisp Machine Lisp. Substantial compatibility with other Lisp dialects (Interlisp, UCILisp, CMULisp) is achieved by means of support packages and compiler switches. The heart of FRANZ LISP is written almost entirely in the programming language C. Of course, it has been greatly extended by additions written in Lisp. A small part is written in the assembly language for the current host machines, VAXen and a couple of flavors of 68000. Because FRANZ LISP is written in C, it is relatively portable and easy to comprehend.

FRANZ LISP is capable of running large lisp programs in a timesharing environment, has facilities for arrays and user defined structures, has a user controlled reader with character and word macro capabilities, and can interact directly with compiled Lisp, C, Fortran, and Pascal code.

This document is a reference manual for the FRANZ LISP system. It is not a Lisp primer or introduction to the language. Some parts will be of interest primarily to those maintaining FRANZ LISP at their computer site. There is an additional document entitled *The Franz Lisp System*, by John Foderaro, which partially describes the system implementation. FRANZ LISP, as delivered by Berkeley, includes all source code and machine readable version of this manual and system document. The system document is in a single file named "franz.n" in the "doc" subdirectory.

This document is divided into four Movements. In the first one we will attempt to describe the language of FRANZ LISP precisely and completely as it now stands (Opus 38.69, June 1983). In the second Movement we will look at the reader, function types, arrays and exception handling. In the third Movement we will look at several large support packages written to help the FRANZ LISP user, namely the trace package, compiler, fixit and stepping package. Finally the fourth movement contains an index into the other movements. In the rest of this chapter we shall examine the data types of FRANZ LISP. The conventions used in the description of the FRANZ LISP functions will be given in §1.3 – it is very important that these conventions are understood.

[†]It is rumored that this name has something to do with Franz Liszt [Frants List] (1811-1886) a Hungarian composer and keyboard virtuoso. These allegations have never been proven.

1.2. Data Types FRANZ LISP has fourteen data types. In this section we shall look in detail at each type and if a type is divisible we shall look inside it. There is a Lisp function *type* which will return the type name of a lisp object. This is the official FRANZ LISP name for that type and we will use this name and this name only in the manual to avoid confusing the reader. The types are listed in terms of importance rather than alphabetically.

1.2.0. lispval This is the name we use to describe any Lisp object. The function *type* will never return 'lispval'.

1.2.1. symbol This object corresponds to a variable in most other programming languages. It may have a value or may be 'unbound'. A symbol may be *lambda bound* meaning that its current value is stored away somewhere and the symbol is given a new value for the duration of a certain context. When the Lisp processor leaves that context, the symbol's current value is thrown away and its old value is restored.

A symbol may also have a *function binding.* This function binding is static; it cannot be lambda bound. Whenever the symbol is used in the functional position of a Lisp expression the function binding of the symbol is examined (see Chapter 4 for more details on evaluation).

A symbol may also have a *property list*, another static data structure. The property list consists of a list of an even number of elements, considered to be grouped as pairs. The first element of the pair is the *indicator* the second the *value* of that indicator.

Each symbol has a print name *(pname)* which is how this symbol is accessed from input and referred to on (printed) output.

A symbol also has a hashlink used to link symbols together in the oblist -- this field is inaccessible to the lisp user.

Symbols are created by the reader and by the functions *concat*, *maknam* and their derivatives. Most symbols live on FRANZ LISP's sole *oblist*, and therefore two symbols with the same print name are usually the exact same object (they are *eq*). Symbols which are not on the oblist are said to be *uninterned.* The function *maknam* creates uninterned symbols while *concat* creates *interned* ones.

Subpart name	Get value	Set value	Type
value	eval	set setq	lispval
property list	plist get	setplist putprop defprop	list or nil
function binding	getd	putd def	array, binary, list or nil
print name	get_pname		string
hash link			

1.2.2. list A list cell has two parts, called the car and cdr. List cells are created by the function *cons*.

Subpart name	Get value	Set value	Type
car	car	rplaca	lispval
cdr	cdr	rplacd	lispval

1.2.3. binary This type acts as a function header for machine coded functions. It has two parts, a pointer to the start of the function and a symbol whose print name describes the argument *discipline*. The discipline (if *lambda*, *macro* or *nlambda*) determines whether the arguments to this function will be evaluated by the caller before this function is called. If the discipline is a string (specifically "*subroutine*", "*function*", "*integer-function*", "*real-function*", "*c-function*", "*double-c-function*", or "*vector-c-function*") then this function is a foreign subroutine or function (see §8.5 for more details on this). Although the type of the *entry* field of a binary type object is usually **string** or **other**, the object pointed to is actually a sequence of machine instructions.
Objects of type binary are created by *mfunction*, *cfasl*, and *getaddress*.

Subpart name	Get value	Set value	Type
entry	getentry		string or fixnum
discipline	getdisc	putdisc	symbol or fixnum

1.2.4. fixnum A fixnum is an integer constant in the range -2^{31} to $2^{31}-1$. Small fixnums (-1024 to 1023) are stored in a special table so they needn't be allocated each time one is needed. In principle, the range for fixnums is machine dependent, although all current implementations for franz have this range.

1.2.5. flonum A flonum is a double precision real number. On the VAX, the range is $\pm 2.9 \times 10^{-37}$ to $\pm 1.7 \times 10^{38}$. There are approximately sixteen decimal digits of precision. Other machines may have other ranges.

1.2.6. bignum A bignum is an integer of potentially unbounded size. When integer arithmetic exceeds the limits of fixnums mentioned above, the calculation is automatically done with bignums. Should calculation with bignums give a result which can be represented as a fixnum, then the fixnum representation will be used[†]. This contraction is known as *integer normalization*. Many Lisp functions assume that integers are normalized. Bignums are composed of a sequence of list cells and a cell known as an **sdot**. The user should consider a **bignum** structure indivisible and use functions such as *haipart*, and *bignum-leftshift* to extract parts of it.

1.2.7. string A string is a null terminated sequence of characters. Most functions of symbols which operate on the symbol's print name will also work on strings. The default reader syntax is set so that a sequence of characters surrounded by double quotes is a string.

1.2.8. port A port is a structure which the system I/O routines can reference to transfer data between the Lisp system and external media. Unlike other Lisp objects there are a very limited number of ports (20). Ports are allocated by *infile* and *outfile* and deallocated by *close* and *resetio*. The *print* function prints a port as a percent sign followed by the name of the file it is connected to (if the port was opened by *fileopen, infile, or outfile*). During initialization, FRANZ LISP binds the symbol **piport** to a port attached to the standard input stream. This port prints as %$stdin. There are ports connected to the standard output and error streams, which print as %$stdout and %$stderr. This is discussed in more detail at the beginning of Chapter 5.

1.2.9. vector Vectors are indexed sequences of data. They can be used to implement a notion of user-defined types via their associated property list. They make **hunks** (see below) logically unnecessary, although hunks are very efficiently garbage collected. There is a second kind of vector, called an immediate-vector, which stores binary data. The name that the function *type* returns for immediate-vectors is **vectori**. Immediate-vectors could be used to implement strings and block-flonum arrays, for example. Vectors are discussed in chapter 9. The functions *new-vector*, and *vector*, can be used to create vectors.

[†]The current algorithms for integer arithmetic operations will return (in certain cases) a result between $\pm 2^{30}$ and 2^{31} as a bignum although this could be represented as a fixnum.

Subpart name	Get value	Set value	Type
datum[*i*]	vref	vset	lispval
property	vprop	vsetprop vputprop	lispval
size	vsize	–	fixnum

1.2.10. array Arrays are rather complicated types and are fully described in Chapter 9. An array consists of a block of contiguous data, a function to access that data, and auxiliary fields for use by the accessing function. Since an array's accessing function is created by the user, an array can have any form the user chooses (e.g. n-dimensional, triangular, or hash table).
Arrays are created by the function *marray*.

Subpart name	Get value	Set value	Type
access function	getaccess	putaccess	binary, list or symbol
auxiliary	getaux	putaux	lispval
data	arrayref	replace set	block of contiguous lispval
length	getlength	putlength	fixnum
delta	getdelta	putdelta	fixnum

1.2.11. value A value cell contains a pointer to a lispval. This type is used mainly by arrays of general lisp objects. Value cells are created with the *ptr* function. A value cell containing a pointer to the symbol 'foo' is printed as '(ptr to)foo'

1.2.12. hunk A hunk is a vector of from 1 to 128 lispvals. Once a hunk is created (by *hunk* or *makhunk*) it cannot grow or shrink. The access time for an element of a hunk is slower than a list cell element but faster than an array. Hunks are really only allocated in sizes which are powers of two, but can appear to the user to be any size in the 1 to 128 range. Users of hunks must realize that *(not (atom 'lispval))* will return true if *lispval* is a hunk. Most lisp systems do not have a direct test for a list cell and instead use the above test and assume that a true result means *lispval* is a list cell. In FRANZ LISP you can use *dtpr* to check for a list cell. Although hunks are not list cells, you can still access the first two hunk elements with *cdr* and *car* and you can access any hunk element with *cxr*[†]. You can set the value of the first two elements of a hunk with *rplacd* and *rplaca* and you can set the value of any element of the hunk with *rplacx*. A hunk is printed by printing its contents

[†]In a hunk, the function *cdr* references the first element and *car* the second.

surrounded by { and }. However a hunk cannot be read in in this way in the standard lisp system. It is easy to write a reader macro to do this if desired.

1.2.13. other Occasionally, you can obtain a pointer to storage not allocated by the lisp system. One example of this is the entry field of those FRANZ LISP functions written in C. Such objects are classified as of type **other**. Foreign functions which call malloc to allocate their own space, may also inadvertantly create such objects. The garbage collector is supposed to ignore such objects.

1.3. Documentation The conventions used in the following chapters were designed to give a great deal of information in a brief space. The first line of a function description contains the function name in **bold face** and then lists the arguments, if any. The arguments all have names which begin with a letter or letters and an underscore. The letter(s) gives the allowable type(s) for that argument according to this table.

Letter	Allowable type(s)
g	any type
s	symbol (although nil may not be allowed)
t	string
l	list (although nil may be allowed)
n	number (fixnum, flonum, bignum)
i	integer (fixnum, bignum)
x	fixnum
b	bignum
f	flonum
u	function type (either binary or lambda body)
y	binary
v	vector
V	vectori
a	array
e	value
p	port (or nil)
h	hunk

In the first line of a function description, those arguments preceded by a quote mark are evaluated (usually before the function is called). The quoting convention is used so that we can give a name to the result of evaluating the argument and we can describe the allowable types. If an argument is not quoted it does not mean that that argument will not be evaluated, but rather that if it is evaluated, the time at which it is evaluated will be specifically mentioned in the function description. Optional arguments are surrounded by square brackets. An ellipsis (...) means zero or more occurrences of an argument of the directly preceding type.

CHAPTER 2

Data Structure Access

The following functions allow one to create and manipulate the various types of lisp data structures. Refer to §1.2 for details of the data structures known to FRANZ LISP.

2.1. Lists

The following functions exist for the creation and manipulating of lists. Lists are composed of a linked list of objects called either 'list cells', 'cons cells' or 'dtpr cells'. Lists are normally terminated with the special symbol **nil**. **nil** is both a symbol and a representation for the empty list ().

2.1.1. list creation

(cons 'g_arg1 'g_arg2**)**

RETURNS:a new list cell whose car is g_arg1 and whose cdr is g_arg2.

(xcons 'g_arg1 'g_arg2**)**

EQUIVALENT TO:*(cons 'g_arg2 'g_arg1)*

(ncons 'g_arg**)**

EQUIVALENT TO:*(cons 'g_arg nil)*

(list ['g_arg1 ...]**)**

RETURNS:a list whose elements are the g_arg*i*.

(append 'l_arg1 'l_arg2**)**

RETURNS:a list containing the elements of l_arg1 followed by l_arg2.

NOTE: To generate the result, the top level list cells of l_arg1 are duplicated and the cdr of the last list cell is set to point to l_arg2. Thus this is an expensive operation if l_arg1 is large. See the descriptions of *nconc* and *tconc* for cheaper ways of doing the *append* if the list l_arg1 can be altered.

(append1 'l_arg1 'g_arg2)

RETURNS:a list like l_arg1 with g_arg2 as the last element.

NOTE: this is equivalent to (append 'l_arg1 (list 'g_arg2)).

```
; A common mistake is using append to add one element to the end of a list
-> (append '(a b c d) 'e)
(a b c d . e)
; The user intended to say:
-> (append '(a b c d) '(e))
(a b c d e)
; better is append1
-> (append1 '(a b c d) 'e)
(a b c d e)
```

(quote! [g_qform*i*] ...[! 'g_eform*i*] ... [!! 'l_form*i*] ...)

RETURNS:The list resulting from the splicing and insertion process described below.

NOTE: *quote!* is the complement of the *list* function. *list* forms a list by evaluating each for in the argument list; evaluation is suppressed if the form is *quote*ed. In *quote!,* each form is implicitly *quote*ed. To be evaluated, a form must be preceded by one of the evaluate operations ! and !!. ! g_eform evaluates g_form and the value is inserted in the place of the call; !! l_form evaluates l_form and the value is spliced into the place of the call.

'Splicing in' means that the parentheses surrounding the list are removed as the example below shows. Use of the evaluate operators can occur at any level in a form argument.

Another way to get the effect of the *quote!* function is to use the backquote character macro (see § 8.3.3).

```
(quote! cons ! (cons 1 2) 3) = (cons (1 . 2) 3)
(quote! 1 !! (list 2 3 4) 5) = (1 2 3 4 5)
(setq quoted 'evaled)(quote! ! ((I am  ! quoted))) = ((I am evaled))
(quote! try ! '(this ! one)) = (try (this ! one))
```

(bignum-to-list 'b_arg**)**

RETURNS:A list of the fixnums which are used to represent the bignum.

NOTE: the inverse of this function is *list-to-bignum.*

(list-to-bignum 'l_ints**)**

WHERE: l_ints is a list of fixnums.

RETURNS:a bignum constructed of the given fixnums.

NOTE: the inverse of this function is *bignum-to-list.*

2.1.2. list predicates

(dtpr 'g_arg**)**

RETURNS:t iff g_arg is a list cell.

NOTE: that (dtpr '()) is nil. The name **dtpr** is a contraction for "dotted pair".

(listp 'g_arg**)**

RETURNS:t iff g_arg is a list object or nil.

(tailp 'l_x 'l_y**)**

RETURNS:l_x, if a list cell *eq* to l_x is found by *cdr*ing down l_y zero or more times, nil otherwise.

```
-> (setq x '(a b c d) y (cddr x))
(c d)
-> (and (dtpr x) (listp x))      ; x and y are dtprs and lists
t
-> (dtpr '())                    ; () is the same as nil and is not a dtpr
nil
-> (listp '())                   ; however it is a list
t
-> (tailp y x)
(c d)
```

(length 'l_arg**)**

RETURNS: the number of elements in the top level of list l_arg.

2.1.3. list accessing

(car 'l_arg**)**
(cdr 'l_arg**)**

RETURNS: *cons* cell. (*car* (*cons* x y)) is always x, (*cdr* (*cons* x y)) is always y. In FRANZ LISP, the cdr portion is located first in memory. This is hardly noticeable, and we mention it primarily as a curiosity.

(c..r 'lh_arg**)**

WHERE: the .. represents any positive number of **a**'s and **d**'s.

RETURNS: the result of accessing the list structure in the way determined by the function name. The **a**'s and **d**'s are read from right to left, a **d** directing the access down the cdr part of the list cell and an **a** down the car part.

NOTE: lh_arg may also be nil, and it is guaranteed that the car and cdr of nil is nil. If lh_arg is a hunk, then *(car 'lh_arg)* is the same as *(cxr 1 'lh_arg)* and *(cdr 'lh_arg)* is the same as *(cxr 0 'lh_arg)*.
It is generally hard to read and understand the context of functions with large strings of **a**'s and **d**'s, but these functions are supported by rapid accessing and open-compiling (see Chapter 12).

(nth 'x_index 'l_list**)**

RETURNS: the nth element of l_list, assuming zero-based index. Thus (nth 0 l_list) is the same as (car l_list). *nth* is both a function, and a compiler macro, so that more efficient code might be generated than for *nthelem* (described below).

NOTE: If x_arg1 is non-positive or greater than the length of the list, nil is returned.

(nthcdr 'x_index 'l_list**)**

RETURNS: the result of *cdr*ing down the list l_list x_index times.

NOTE: If x_index is less than 0, then *(cons nil 'l_list)* is returned.

(nthelem 'x_arg1 'l_arg2**)**

RETURNS: The x_arg1'*st* element of the list l_arg2.

NOTE: This function comes from the PDP-11 Lisp system.

(last 'l_arg**)**

RETURNS:the last list cell in the list l_arg.

EXAMPLE:*last* does NOT return the last element of a list!
(last '(a b)) = (b)

(ldiff 'l_x 'l_y**)**

RETURNS:a list of all elements in l_x but not in l_y , i.e., the list difference of l_x and l_y.

NOTE: l_y must be a tail of l_x, i.e., *eq* to the result of applying some number of *cdr*'s to l_x. Note that the value of *ldiff* is always new list structure unless l_y is nil, in which case *(ldiff l_x nil)* is l_x itself. If l_y is not a tail of l_x, *ldiff* generates an error.

EXAMPLE:*(ldiff 'l_x (member 'g_foo 'l_x))* gives all elements in l_x up to the first g_foo.

2.1.4. list manipulation

(rplaca 'lh_arg1 'g_arg2**)**

RETURNS:the modified lh_arg1.

SIDE EFFECT: the car of lh_arg1 is set to g_arg2. If lh_arg1 is a hunk then the second element of the hunk is set to g_arg2.

(rplacd 'lh_arg1 'g_arg2**)**

RETURNS:the modified lh_arg1.

SIDE EFFECT: the cdr of lh_arg2 is set to g_arg2. If lh_arg1 is a hunk then the first element of the hunk is set to g_arg2.

(attach 'g_x 'l_l**)**

RETURNS:l_l whose *car* is now g_x, whose *cadr* is the original *(car l_l)*, and whose *cddr* is the original *(cdr l_l)*.

NOTE: what happens is that g_x is added to the beginning of list l_l yet maintaining the same list cell at the beginning of the list.

(delete 'g_val 'l_list ['x_count]**)**

RETURNS:the result of splicing g_val from the top level of l_list no more than x_count times.

NOTE: x_count defaults to a very large number, thus if x_count is not given, all occurrences of g_val are removed from the top level of l_list. g_val is compared with successive *car*'s of l_list using the function *equal*.

SIDE EFFECT: l_list is modified using rplacd, no new list cells are used.

(delq 'g_val 'l_list ['x_count])
(dremove 'g_val 'l_list ['x_count])

RETURNS: the result of splicing g_val from the top level of l_list no more than x_count times.

NOTE: *delq* (and *dremove*) are the same as *delete* except that *eq* is used for comparison instead of *equal.*

```
; note that you should use the value returned by delete or delq
; and not assume that g_val will always show the deletions.
; For example

-> (setq test '(a b c a d e))
(a b c a d e)
-> (delete 'a test)
(b c d e)       ; the value returned is what we would expect
-> test
(a b c d e)     ; but test still has the first a in the list!
```

(remq 'g_x 'l_l ['x_count])
(remove 'g_x 'l_l)

RETURNS: a *copy* of l_l with all top level elements *equal* to g_x removed. *remq* uses *eq* instead of *equal* for comparisons.

NOTE: remove does not modify its arguments like *delete*, and *delq* do.

(insert 'g_object 'l_list 'u_comparefn 'g_nodups)

RETURNS: a list consisting of l_list with g_object destructively inserted in a place determined by the ordering function u_comparefn.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil if g_x can precede g_y in sorted order, nil if g_y must precede g_x. If u_comparefn is nil, alphabetical order will be used. If g_nodups is non-nil, an element will not be inserted if an equal element is already in the list. *insert* does binary search to determine where to insert the new element.

(merge 'l_data1 'l_data2 'u_comparefn)

RETURNS: the merged list of the two input sorted lists l_data1 and l_data1 using binary comparison function u_comparefn.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil if g_x can precede g_y in sorted order, nil if g_y must precede g_x. If u_comparefn is nil, alphabetical order will be used. u_comparefn should be thought of as "less than or equal". *merge* changes both of its data arguments.

(subst 'g_x 'g_y 'l_s)
(dsubst 'g_x 'g_y 'l_s)

RETURNS: the result of substituting g_x for all *equal* occurrences of g_y at all levels in l_s.

NOTE: If g_y is a symbol, *eq* will be used for comparisons. The function *subst* does not modify l_s but the function *dsubst* (destructive substitution) does.

(lsubst 'l_x 'g_y 'l_s)

RETURNS: a copy of l_s with l_x spliced in for every occurrence of of g_y at all levels. Splicing in means that the parentheses surrounding the list l_x are removed as the example below shows.

```
-> (subst '(a b c) 'x '(x y z (x y z) (x y z)))
((a b c) y z ((a b c) y z) ((a b c) y z))
-> (lsubst '(a b c) 'x '(x y z (x y z) (x y z)))
(a b c y z (a b c y z) (a b c y z))
```

(subpair 'l_old 'l_new 'l_expr)

WHERE: there are the same number of elements in l_old as l_new.

RETURNS: the list l_expr with all occurrences of a object in l_old replaced by the corresponding one in l_new. When a substitution is made, a copy of the value to substitute in is not made.

EXAMPLE: *(subpair '(a c)' (x y) '(a b c d)) = (x b y d)*

(nconc 'l_arg1 'l_arg2 ['l_arg3 ...])

RETURNS: A list consisting of the elements of l_arg1 followed by the elements of l_arg2 followed by l_arg3 and so on.

NOTE: The *cdr* of the last list cell of l_arg*i* is changed to point to l_arg*i+1*.

```
; nconc is faster than append because it doesn't allocate new list cells.
-> (setq lis1 '(a b c))
(a b c)
-> (setq lis2 '(d e f))
(d e f)
-> (append lis1 lis2)
(a b c d e f)
-> lis1
(a b c)      ; note that lis1 has not been changed by append
-> (nconc lis1 lis2)
(a b c d e f) ; nconc returns the same value as append
-> lis1
(a b c d e f) ; but in doing so alters lis1
```

(reverse 'l_arg**)**
(nreverse 'l_arg**)**

RETURNS: the list l_arg with the elements at the top level in reverse order.

NOTE: The function *nreverse* does the reversal in place, that is the list structure is modified.

(nreconc 'l_arg 'g_arg**)**

EQUIVALENT TO: *(nconc (nreverse 'l_arg) 'g_arg)*

2.2. Predicates

The following functions test for properties of data objects. When the result of the test is either 'false' or 'true', then **nil** will be returned for 'false' and something other than **nil** (often **t**) will be returned for 'true'.

(arrayp 'g_arg**)**

RETURNS: t iff g_arg is of type array.

(atom 'g_arg**)**

RETURNS: t iff g_arg is not a list or hunk object.

NOTE: *(atom '())* returns t.

(bcdp 'g_arg**)**

RETURNS:t iff g_arg is a data object of type binary.

NOTE: This function is a throwback to the PDP-11 Lisp system. The name stands for binary code predicate.

(bigp 'g_arg**)**

RETURNS:t iff g_arg is a bignum.

(dtpr 'g_arg**)**

RETURNS:t iff g_arg is a list cell.

NOTE: that (dtpr '()) is nil.

(hunkp 'g_arg**)**

RETURNS:t iff g_arg is a hunk.

(listp 'g_arg**)**

RETURNS:t iff g_arg is a list object or nil.

(stringp 'g_arg**)**

RETURNS:t iff g_arg is a string.

(symbolp 'g_arg**)**

RETURNS:t iff g_arg is a symbol.

(valuep 'g_arg**)**

RETURNS:t iff g_arg is a value cell

(vectorp 'v_vector**)**

RETURNS:t iff the argument is a vector.

(vectorip 'v_vector**)**

RETURNS:t iff the argument is an immediate-vector.

(type 'g_arg**)**
(typep 'g_arg**)**

RETURNS:a symbol whose pname describes the type of g_arg.

(signp s_test 'g_val)

RETURNS:t iff g_val is a number and the given test s_test on g_val returns true.

NOTE: The fact that *signp* simply returns nil if g_val is not a number is probably the most important reason that *signp* is used. The permitted values for s_test and what they mean are given in this table.

s_test	tested
l	g_val < 0
le	g_val ≤ 0
e	g_val = 0
n	g_val ≠ 0
ge	g_val ≥ 0
g	g_val > 0

(eq 'g_arg1 'g_arg2)

RETURNS:t if g_arg1 and g_arg2 are the exact same lisp object.

NOTE: *Eq* simply tests if g_arg1 and g_arg2 are located in the exact same place in memory. Lisp objects which print the same are not necessarily *eq*. ' The only objects guaranteed to be *eq* are interned symbols with the same print name. [Unless a symbol is created in a special way (such as with *uconcat* or *maknam*) it will be interned.]

(neq 'g_x 'g_y)

RETURNS:t if g_x is not *eq* to g_y, otherwise nil.

(equal 'g_arg1 'g_arg2)
(eqstr 'g_arg1 'g_arg2)

RETURNS:t iff g_arg1 and g_arg2 have the same structure as described below.

NOTE: g_arg and g_arg2 are *equal* if

(1) they are *eq*.

(2) they are both fixnums with the same value

(3) they are both flonums with the same value

(4) they are both bignums with the same value

(5) they are both strings and are identical.

(6) they are both lists and their cars and cdrs are *equal*.

```
; eq is much faster than equal, especially in compiled code,
; however you cannot use eq to test for equality of numbers outside
; of the range -1024 to 1023.  equal will always work.
-> (eq 1023 1023)
t
-> (eq 1024 1024)
nil
-> (equal 1024 1024)
t
```

(not 'g_arg**)**
(null 'g_arg**)**

RETURNS:t iff g_arg is nil.

(member 'g_arg1 'l_arg2**)**
(memq 'g_arg1 'l_arg2**)**

RETURNS:that part of the l_arg2 beginning with the first occurrence of g_arg1. If g_arg1 is not in the top level of l_arg2, nil is returned.

NOTE: *member* tests for equality with *equal*, *memq* tests for equality with *eq*.

2.3. Symbols and Strings

In many of the following functions the distinction between symbols and strings is somewhat blurred. To remind ourselves of the difference, a string is a null terminated sequence of characters, stored as compactly as possible. Strings are used as constants in FRANZ LISP. They *eval* to themselves. A symbol has additional structure: a value, property list, function binding, as well as its external representation (or print-name). If a symbol is given to one of the string manipulation functions below, its print name will be used as the string.

Another popular way to represent strings in Lisp is as a list of fixnums which represent characters. The suffix 'n' to a string manipulation function indicates that it returns a string in this form.

2.3.1. symbol and string creation

(concat ['stn_arg1 ...])
(uconcat ['stn_arg1 ...])

RETURNS: a symbol whose print name is the result of concatenating the print names, string characters or numerical representations of the sn_arg*i*.

NOTE: If no arguments are given, a symbol with a null pname is returned. *concat* places the symbol created on the oblist, the function *uconcat* does the same thing but does not place the new symbol on the oblist.

EXAMPLE: *(concat 'abc (add 3 4) "def")* = abc7def

(concatl 'l_arg)

EQUIVALENT TO: *(apply 'concat 'l_arg)*

(implode 'l_arg)
(maknam 'l_arg)

WHERE: l_arg is a list of symbols, strings and small fixnums.

RETURNS: The symbol whose print name is the result of concatenating the first characters of the print names of the symbols and strings in the list. Any fixnums are converted to the equivalent ascii character. In order to concatenate entire strings or print names, use the function *concat*.

NOTE: *implode* interns the symbol it creates, *maknam* does not.

(gensym ['s_leader])

RETURNS: a new uninterned atom beginning with the first character of s_leader's pname, or beginning with g if s_leader is not given.

NOTE: The symbol looks like x0nnnnn where x is s_leader's first character and nnnnn is the number of times you have called gensym.

(copysymbol 's_arg 'g_pred)

RETURNS: an uninterned symbol with the same print name as s_arg. If g_pred is non nil, then the value, function binding and property list of the new symbol are made *eq* to those of s_arg.

(ascii 'x_charnum)

WHERE: x_charnum is between 0 and 255.

RETURNS: a symbol whose print name is the single character whose fixnum representation is x_charnum.

(intern 's_arg**)**

RETURNS:s_arg

SIDE EFFECT: s_arg is put on the oblist if it is not already there.

(remob 's_symbol**)**

RETURNS:s_symbol

SIDE EFFECT: s_symbol is removed from the oblist.

(rematom 's_arg**)**

RETURNS:t if s_arg is indeed an atom.

SIDE EFFECT: s_arg is put on the free atoms list, effectively reclaiming an atom cell.

NOTE: This function does *not* check to see if s_arg is on the oblist or is referenced anywhere. Thus calling *rematom* on an atom in the oblist may result in disaster when that atom cell is reused!

2.3.2. string and symbol predicates

(boundp 's_name**)**

RETURNS:nil if s_name is unbound: that is, it has never been given a value. If x_name has the value g_val, then (nil . g_val) is returned. See also *makunbound.*

(alphalessp 'st_arg1 'st_arg2**)**

RETURNS:t iff the 'name' of st_arg1 is alphabetically less than the name of st_arg2. If st_arg is a symbol then its 'name' is its print name. If st_arg is a string, then its 'name' is the string itself.

2.3.3. symbol and string accessing

(symeval 's_arg**)**

RETURNS:the value of symbol s_arg.

NOTE: It is illegal to ask for the value of an unbound symbol. This function has the same effect as *eval*, but compiles into much more efficient code.

(get_pname 's_arg**)**

RETURNS:the string which is the print name of s_arg.

(plist 's_arg**)**

RETURNS:the property list of s_arg.

(getd 's_arg**)**

RETURNS:the function definition of s_arg or nil if there is no function definition.

NOTE: the function definition may turn out to be an array header.

(getchar 's_arg 'x_index**)**
(nthchar 's_arg 'x_index**)**
(getcharn 's_arg 'x_index**)**

RETURNS:the x_index*th* character of the print name of s_arg or nil if x_index is less than 1 or greater than the length of s_arg's print name.

NOTE: *getchar* and *nthchar* return a symbol with a single character print name, *getcharn* returns the fixnum representation of the character.

(substring 'st_string 'x_index ['x_length]**)**
(substringn 'st_string 'x_index ['x_length]**)**

RETURNS:a string of length at most x_length starting at x_index*th* character in the string.

NOTE: If x_length is not given, all of the characters for x_index to the end of the string are returned. If x_index is negative the string begins at the x_index*th* character from the end. If x_index is out of bounds, nil is returned.

NOTE: *substring* returns a list of symbols, *substringn* returns a list of fixnums. If *substringn* is given a 0 x_length argument then a single fixnum which is the x_index*th* character is returned.

2.3.4. symbol and string manipulation

(set 's_arg1 'g_arg2**)**

RETURNS:g_arg2.

SIDE EFFECT: the value of s_arg1 is set to g_arg2.

(setq s_atm1 'g_val1 [s_atm2 'g_val2 ... ...]**)**

WHERE: the arguments are pairs of atom names and expressions.

RETURNS:the last g_val*i*.

SIDE EFFECT: each s_atm*i* is set to have the value g_val*i*.

NOTE: *set* evaluates all of its arguments, *setq* does not evaluate the s_atm*i*.

(desetq sl_pattern1 'g_exp1 [... ...])

RETURNS:g_expn

SIDE EFFECT: This acts just like *setq* if all the sl_pattern*i* are symbols. If sl_pattern*i* is a list then it is a template which should have the same structure as g_exp*i* The symbols in sl_pattern are assigned to the corresponding parts of g_exp. (See also *setf*)

EXAMPLE:*(desetq (a b (c . d)) '(1 2 (3 4 5)))*
sets a to 1, b to 2, c to 3, and d to (4 5).

(setplist 's_atm 'l_plist)

RETURNS:l_plist.

SIDE EFFECT: the property list of s_atm is set to l_plist.

(makunbound 's_arg)

RETURNS:s_arg

SIDE EFFECT: the value of s_arg is made 'unbound'. If the interpreter attempts to evaluate s_arg before it is again given a value, an unbound variable error will occur.

(aexplode 's_arg)
(explode 'g_arg)
(aexplodec 's_arg)
(explodec 'g_arg)
(aexploden 's_arg)
(exploden 'g_arg)

RETURNS:a list of the characters used to print out s_arg or g_arg.

NOTE: The functions beginning with 'a' are internal functions which are limited to symbol arguments. The functions *aexplode* and *explode* return a list of characters which *print* would use to print the argument. These characters include all necessary escape characters. Functions *aexplodec* and *explodec* return a list of characters which *patom* would use to print the argument (i.e. no escape characters). Functions *aexploden* and *exploden* are similar to *aexplodec* and *explodec* except that a list of fixnum equivalents of characters are returned.

```
-> (setq x '|quote this \| ok?|)
|quote this \| ok?|
-> (explode x)
(q u o t e |\\| | | t h i s |\\| | | |\\| |\|| |\\| | | o k ?)
; note that |\\| just means the single character: backslash.
; and |\|| just means the single character: vertical bar
; and | | means the single character: space

-> (explodec x)
(q u o t e | | t h i s | | |\|| | | o k ?)
-> (exploden x)
(113 117 111 116 101 32 116 104 105 115 32 124 32 111 107 63)
```

2.4. Vectors

See Chapter 9 for a discussion of vectors. They are less efficient that hunks but more efficient than arrays.

2.4.1. vector creation

(new-vector 'x_size ['g_fill ['g_prop]])

RETURNS: A **vector** of length x_size. Each data entry is initialized to g_fill, or to nil, if the argument g_fill is not present. The vector's property is set to g_prop, or to nil, by default.

(new-vectori-byte 'x_size ['g_fill ['g_prop]])
(new-vectori-word 'x_size ['g_fill ['g_prop]])
(new-vectori-long 'x_size ['g_fill ['g_prop]])

RETURNS: A **vectori** with x_size elements in it. The actual memory requirement is two long words + x_size*(n bytes), where n is 1 for new-vector-byte, 2 for new-vector-word, or 4 for new-vectori-long. Each data entry is initialized to g_fill, or to zero, if the argument g_fill is not present. The vector's property is set to g_prop, or nil, by default.

Vectors may be created by specifying multiple initial values:

(vector ['g_val0 'g_val1 ...])

RETURNS: a **vector**, with as many data elements as there are arguments. It is quite possible to have a vector with no data elements. The vector's property will be a null list.

(vectori-byte ['x_val0 'x_val2 ...])
(vectori-word ['x_val0 'x_val2 ...])
(vectori-long ['x_val0 'x_val2 ...])

RETURNS: a **vectori**, with as many data elements as there are arguments. The arguments are required to be fixnums. Only the low order byte or word is used in the case of vectori-byte and vectori-word. The vector's property will be null.

2.4.2. vector reference

(vref 'v_vect 'x_index)
(vrefi-byte 'V_vect 'x_bindex)
(vrefi-word 'V_vect 'x_windex)
(vrefi-long 'V_vect 'x_lindex)

RETURNS: the desired data element from a vector. The indices must be fixnums. Indexing is zero-based. The vrefi functions sign extend the data.

(vprop 'Vv_vect)

RETURNS: The Lisp property associated with a vector.

(vget 'Vv_vect 'g_ind)

RETURNS: The value stored under g_ind if the Lisp property associated with 'Vv_vect is a disembodied property list.

(vsize 'Vv_vect)
(vsize-byte 'V_vect)
(vsize-word 'V_vect)

RETURNS: the number of data elements in the vector. For immediate-vectors, the functions vsize-byte and vsize-word return the number of data elements, if one thinks of the binary data as being comprised of bytes or words.

2.4.3. vector modfication

(vset 'v_vect 'x_index 'g_val)
(vseti-byte 'V_vect 'x_bindex 'x_val)
(vseti-word 'V_vect 'x_windex 'x_val)
(vseti-long 'V_vect 'x_lindex 'x_val)

RETURNS: the datum.

SIDE EFFECT: The indexed element of the vector is set to the value. As noted above, for vseti-word and vseti-byte, the index is construed as the number of the data element within the vector. It is not a byte address. Also, for those two functions, the low order byte or word of x_val is what is stored.

(vsetprop 'Vv_vect 'g_value)

RETURNS: g_value. This should be either a symbol or a disembodied property list whose *car* is a symbol identifying the type of the vector.

SIDE EFFECT: the property list of Vv_vect is set to g_value.

(vputprop 'Vv_vect 'g_value 'g_ind)

RETURNS: g_value.

SIDE EFFECT: If the vector property of Vv_vect is a disembodied property list, then vputprop adds the value g_value under the indicator g_ind. Otherwise, the old vector property is made the first element of the list.

2.5. Arrays

See Chapter 9 for a complete description of arrays. Some of these functions are part of a Maclisp array compatibility package representing only one simple way of using the array structure of FRANZ LISP.

2.5.1. array creation

(marray 'g_data 's_access 'g_aux 'x_length 'x_delta)

RETURNS: an array type with the fields set up from the above arguments in the obvious way (see § 1.2.10).

(*array 's_name 's_type 'x_dim1 ... 'x_dim*n*)
(array s_name s_type x_dim1 ... x_dim*n*)

WHERE: s_type may be one of t, nil, fixnum, flonum, fixnum-block and flonum-block.

RETURNS: an array of type s_type with n dimensions of extents given by the x_dim*i*.

SIDE EFFECT: If s_name is non nil, the function definition of s_name is set to the array structure returned.

NOTE: These functions create a Maclisp compatible array. In FRANZ LISP arrays of type t, nil, fixnum and flonum are equivalent and the elements of these arrays can be any type of lisp object. Fixnum-block and flonum-block arrays are restricted to fixnums and flonums respectively and are used mainly to communicate with foreign functions (see §8.5).

NOTE: ******array* evaluates its arguments, *array* does not.

2.5.2. array predicate

(arrayp 'g_arg**)**

RETURNS:t iff g_arg is of type array.

2.5.3. array accessors

(getaccess 'a_array**)**
(getaux 'a_array**)**
(getdelta 'a_array**)**
(getdata 'a_array**)**
(getlength 'a_array**)**

RETURNS:the field of the array object a_array given by the function name.

(arrayref 'a_name 'x_ind**)**

RETURNS:the x_ind*th* element of the array object a_name. x_ind of zero accesses the first element.

NOTE: *arrayref* uses the data, length and delta fields of a_name to determine which object to return.

(arraycall s_type 'as_array 'x_ind1 ... **)**

RETURNS:the element selected by the indices from the array a_array of type s_type.

NOTE: If as_array is a symbol then the function binding of this symbol should contain an array object.
s_type is ignored by *arraycall* but is included for compatibility with Maclisp.

(arraydims 's_name**)**

RETURNS:a list of the type and bounds of the array s_name.

(listarray 'sa_array ['x_elements]**)**

RETURNS:a list of all of the elements in array sa_array. If x_elements is given, then only the first x_elements are returned.

```
; We will create a 3 by 4 array of general lisp objects
-> (array ernie t 3 4)
array[12]

; the array header is stored in the function definition slot of the
; symbol ernie
-> (arrayp (getd 'ernie))
t
-> (arraydims (getd 'ernie))
(t 3 4)

; store in ernie[2][2] the list (test list)
-> (store (ernie 2 2) '(test list))
(test list)

; check to see if it is there
-> (ernie 2 2)
(test list)

; now use the low level function arrayref to find the same element
; arrays are 0 based and row-major (the last subscript varies the fastest)
; thus element [2][2] is the 10th element , (starting at 0).
-> (arrayref (getd 'ernie) 10)
(ptr to)(test list)    ; the result is a value cell (thus the (ptr to))
```

2.5.4. array manipulation

(putaccess 'a_array 'su_func)
(putaux 'a_array 'g_aux)
(putdata 'a_array 'g_arg)
(putdelta 'a_array 'x_delta)
(putlength 'a_array 'x_length)

RETURNS: the second argument to the function.

SIDE EFFECT: The field of the array object given by the function name is replaced by the second argument to the function.

(store 'l_arexp 'g_val)

WHERE: l_arexp is an expression which references an array element.

RETURNS: g_val

SIDE EFFECT: the array location which contains the element which l_arexp references is changed to contain g_val.

(fillarray 's_array 'l_itms**)**

RETURNS:s_array

SIDE EFFECT: the array s_array is filled with elements from l_itms. If there are not enough elements in l_itms to fill the entire array, then the last element of l_itms is used to fill the remaining parts of the array.

2.6. Hunks

Hunks are vector-like objects whose size can range from 1 to 128 elements. Internally, hunks are allocated in sizes which are powers of 2. In order to create hunks of a given size, a hunk with at least that many elements is allocated and a distinguished symbol EMPTY is placed in those elements not requested. Most hunk functions respect those distinguished symbols, but there are two *(*makhunk* and **rplacx)* which will overwrite the distinguished symbol.

2.6.1. hunk creation

(hunk 'g_val1 ['g_val2 ... 'g_val*n*])

RETURNS:a hunk of length n whose elements are initialized to the g_val*i*.

NOTE: the maximum size of a hunk is 128.

EXAMPLE:*(hunk 4 'sharp 'keys)* = {4 sharp keys}

(makhunk 'xl_arg**)**

RETURNS:a hunk of length xl_arg initialized to all nils if xl_arg is a fixnum. If xl_arg is a list, then we return a hunk of size *(length 'xl_arg)* initialized to the elements in xl_arg.

NOTE: *(makhunk '(a b c))* is equivalent to *(hunk 'a 'b 'c)*.

EXAMPLE:*(makhunk 4)* = {*nil nil nil nil*}

(*makhunk 'x_arg**)**

RETURNS:a hunk of size 2^{x_arg} initialized to EMPTY.

NOTE: This is only to be used by such functions as *hunk* and *makhunk* which create and initialize hunks for users.

2.6.2. hunk accessor

(cxr 'x_ind 'h_hunk**)**

RETURNS:element x_ind (starting at 0) of hunk h_hunk.

(hunk-to-list 'h_hunk**)**

RETURNS:a list consisting of the elements of h_hunk.

2.6.3. hunk manipulators

(rplacx 'x_ind 'h_hunk 'g_val**)**
(*rplacx 'x_ind 'h_hunk 'g_val**)**

RETURNS:h_hunk

SIDE EFFECT: Element x_ind (starting at 0) of h_hunk is set to g_val.

NOTE: *rplacx* will not modify one of the distinguished (EMPTY) elements whereas **rplacx* will.

(hunksize 'h_arg**)**

RETURNS:the size of the hunk h_arg.

EXAMPLE:*(hunksize (hunk 1 2 3))* = 3

2.7. Bcds

A bcd object contains a pointer to compiled code and to the type of function object the compiled code represents.

(getdisc 'y_bcd**)**
(getentry 'y_bcd**)**

RETURNS:the field of the bcd object given by the function name.

(putdisc 'y_func 's_discipline**)**

RETURNS:s_discipline

SIDE EFFECT: Sets the discipline field of y_func to s_discipline.

2.8. Structures

There are three common structures constructed out of list cells: the assoc list, the property list and the tconc list. The functions below manipulate these structures.

2.8.1. assoc list

An 'assoc list' (or alist) is a common lisp data structure. It has the form
((key1 . value1) (key2 . value2) (key3 . value3) ... (keyn . valuen))

(assoc 'g_arg1 'l_arg2**)**
(assq 'g_arg1 'l_arg2**)**

RETURNS: the first top level element of l_arg2 whose *car* is *equal* (with *assoc*) or *eq* (with *assq*) to g_arg1.

NOTE: Usually l_arg2 has an *a-list* structure and g_arg1 acts as key.

(sassoc 'g_arg1 'l_arg2 'sl_func**)**

RETURNS: the result of *(cond ((assoc 'g_arg 'l_arg2) (apply 'sl_func nil)))*

NOTE: sassoc is written as a macro.

(sassq 'g_arg1 'l_arg2 'sl_func**)**

RETURNS: the result of *(cond ((assq 'g_arg 'l_arg2) (apply 'sl_func nil)))*

NOTE: sassq is written as a macro.

```
; assoc or assq is given a key and an assoc list and returns
; the key and value item if it exists, they differ only in how they test
; for equality of the keys.

-> (setq alist '((alpha . a) ( (complex key) . b) (junk . x)))
((alpha . a) ((complex key) . b) (junk . x))

; we should use assq when the key is an atom
-> (assq 'alpha alist)
(alpha . a)

; but it may not work when the key is a list
-> (assq '(complex key) alist)
nil

; however assoc will always work
-> (assoc '(complex key) alist)
((complex key) . b)
```

(sublis 'l_alst 'l_exp**)**

WHERE: l_alst is an *a-list*.

RETURNS: the list l_exp with every occurrence of key*i* replaced by val*i*.

NOTE: new list structure is returned to prevent modification of l_exp. When a substitution is made, a copy of the value to substitute in is not made.

2.8.2. property list

A property list consists of an alternating sequence of keys and values. Normally a property list is stored on a symbol. A list is a 'disembodied' property list if it contains an odd number of elements, the first of which is ignored.

(plist 's_name)

RETURNS: the property list of s_name.

(setplist 's_atm 'l_plist)

RETURNS: l_plist.

SIDE EFFECT: the property list of s_atm is set to l_plist.

(get 'ls_name 'g_ind)

RETURNS: the value under indicator g_ind in ls_name's property list if ls_name is a symbol.

NOTE: If there is no indicator g_ind in ls_name's property list nil is returned. If ls_name is a list of an odd number of elements then it is a disembodied property list. *get* searches a disembodied property list by starting at its *cdr*, and comparing every other element with g_ind, using *eq*.

(getl 'ls_name 'l_indicators)

RETURNS: the property list ls_name beginning at the first indicator which is a member of the list l_indicators, or nil if none of the indicators in l_indicators are on ls_name's property list.

NOTE: If ls_name is a list, then it is assumed to be a disembodied property list.

(putprop 'ls_name 'g_val 'g_ind)
(defprop ls_name g_val g_ind)

RETURNS: g_val.

SIDE EFFECT: Adds to the property list of ls_name the value g_val under the indicator g_ind.

NOTE: *putprop* evaluates it arguments, *defprop* does not. ls_name may be a disembodied property list, see *get*.

(remprop 'ls_name 'g_ind)

RETURNS: the portion of ls_name's property list beginning with the property under the indicator g_ind. If there is no g_ind indicator in ls_name's plist, nil is returned.

SIDE EFFECT: the value under indicator g_ind and g_ind itself is removed from the property list of ls_name.

NOTE: ls_name may be a disembodied property list, see *get*.

```
-> (putprop 'xlate 'a 'alpha)
a
-> (putprop 'xlate 'b 'beta)
b
-> (plist 'xlate)
(alpha a beta b)
-> (get 'xlate 'alpha)
a
; use of a disembodied property list:
-> (get '(nil fateman rjf sklower kls foderaro jkf) 'sklower)
kls
```

2.8.3. tconc structure

A tconc structure is a special type of list designed to make it easy to add objects to the end. It consists of a list cell whose *car* points to a list of the elements added with *tconc* or *lconc* and whose *cdr* points to the last list cell of the list pointed to by the *car*.

(tconc 'l_ptr 'g_x**)**

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with g_x added to the end.

(lconc 'l_ptr 'l_x**)**

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with the list l_x spliced in at the end.

```
; A tconc structure can be initialized in two  ways.
; nil can be given to tconc in which case tconc will generate
; a tconc structure.

->(setq foo (tconc nil 1))
((1) 1)

; Since tconc destructively adds to
; the list, you can now add to foo without using setq again.

->(tconc foo 2)
((1 2) 2)
->foo
((1 2) 2)

; Another way to create a null  tconc structure
; is to use (ncons nil).

->(setq foo (ncons nil))
(nil)
->(tconc foo 1)
((1) 1)

; now see what lconc can do
-> (lconc foo nil)
((1) 1)                          ; no change
-> (lconc foo '(2 3 4))
((1 2 3 4) 4)
```

2.8.4. fclosures

An fclosure is a functional object which admits some data manipulations. They are discussed in §8.4. Internally, they are constructed from vectors.

(fclosure 'l_vars 'g_funobj)

WHERE: l_vars is a list of variables, g_funobj is any object that can be funcalled (including, fclosures).

RETURNS: A vector which is the fclosure.

(fclosure-alist 'v_fclosure**)**

RETURNS: An association list representing the variables in the fclosure. This is a snapshot of the current state of the fclosure. If the bindings in the fclosure are changed, any previously calculated results of *fclosure-alist* will not change.

(fclosure-function 'v_fclosure**)**

RETURNS: the functional object part of the fclosure.

(fclosurep 'v_fclosure**)**

RETURNS: t iff the argument is an fclosure.

(symeval-in-fclosure 'v_fclosure 's_symbol**)**

RETURNS: the current binding of a particular symbol in an fclosure.

(set-in-fclosure 'v_fclosure 's_symbol 'g_newvalue**)**

RETURNS: g_newvalue.

SIDE EFFECT: The variable s_symbol is bound in the fclosure to g_newvalue.

2.9. Random functions

The following functions don't fall into any of the classifications above.

(bcdad 's_funcname**)**

RETURNS: a fixnum which is the address in memory where the function s_funcname begins. If s_funcname is not a machine coded function (binary) then *bcdad* returns nil.

(copy 'g_arg**)**

RETURNS: A structure *equal* to g_arg but with new list cells.

(copyint* 'x_arg**)**

RETURNS: a fixnum with the same value as x_arg but in a freshly allocated cell.

(cpy1 'xvt_arg**)**

RETURNS: a new cell of the same type as xvt_arg with the same value as xvt_arg.

(getaddress 's_entry1 's_binder1 'st_discipline1 [... ... ...]**)**

RETURNS: the binary object which s_binder1's function field is set to.

NOTE: This looks in the running lisp's symbol table for a symbol with the same name as s_entry*i*. It then creates a binary object whose entry field points to s_entry*i* and whose discipline is st_discipline*i*. This binary object is stored in the function field of s_binder*i*. If st_discipline*i* is nil, then "subroutine" is used by default. This is especially useful for *cfasl* users.

(macroexpand 'g_form)

RETURNS:g_form after all macros in it are expanded.

NOTE: This function will only macroexpand expressions which could be evaluated and it does not know about the special nlambdas such as *cond* and *do*, thus it misses many macro expansions.

(ptr 'g_arg)

RETURNS:a value cell initialized to point to g_arg.

(quote g_arg)

RETURNS:g_arg.

NOTE: the reader allows you to abbreviate (quote foo) as 'foo.

(kwote 'g_arg)

RETURNS: *(list (quote quote) g_arg)*.

(replace 'g_arg1 'g_arg2)

WHERE: g_arg1 and g_arg2 must be the same type of lispval and not symbols or hunks.

RETURNS:g_arg2.

SIDE EFFECT: The effect of *replace* is dependent on the type of the g_arg*i* although one will notice a similarity in the effects. To understand what *replace* does to fixnum and flonum arguments, you must first understand that such numbers are 'boxed' in FRANZ LISP. What this means is that if the symbol x has a value 32412, then in memory the value element of x's symbol structure contains the address of another word of memory (called a box) with 32412 in it.

Thus, there are two ways of changing the value of x: the first is to change the value element of x's symbol structure to point to a word of memory with a different value. The second way is to change the value in the box which x points to. The former method is used almost all of the time, the latter is used very rarely and has the potential to cause great confusion. The function *replace* allows you to do the latter, i.e., to actually change the value in the box.

You should watch out for these situations. If you do *(setq y x)*, then both x and y will point to the same box. If you now *(replace x 12345)*, then y will also have the value 12345. And, in fact, there may be many other pointers to that box.

Another problem with replacing fixnums is that some boxes are read-only. The fixnums between -1024 and 1023 are stored in a read-only area and attempts to replace them will result in an "Illegal memory reference" error (see the description of *copyint** for a way around this problem).

For the other valid types, the effect of *replace* is easy to understand. The fields of g_val1's structure are made eq to the corresponding fields of g_val2's structure. For example, if x and y have lists as values then the effect of *(replace x y)* is the same as *(rplaca x (car y))* and *(rplacd x (cdr y))*.

(scons 'x_arg 'bs_rest)

WHERE: bs_rest is a bignum or nil.

RETURNS: a bignum whose first bigit is x_arg and whose higher order bigits are bs_rest.

(setf g_refexpr 'g_value)

NOTE: *setf* is a generalization of setq. Information may be stored by binding variables, replacing entries of arrays, and vectors, or being put on property lists, among others. Setf will allow the user to store data into some location, by mentioning the operation used to refer to the location. Thus, the first argument may be partially evaluated, but only to the extent needed to calculate a reference. *setf* returns g_value. (Compare to *desetq*)

```
(setf x 3)       = (setq x 3)
(setf (car x) 3) = (rplaca x 3)
(setf (get foo 'bar) 3) = (putprop foo 3 'bar)
(setf (vref vector index) value) = (vset vector index value)
```

(sort 'l_data 'u_comparefn)

RETURNS: a list of the elements of l_data ordered by the comparison function u_comparefn.

SIDE EFFECT: the list l_data is modified rather than allocated in new storage.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil if g_x can precede g_y in sorted order; nil if g_y must precede g_x. If u_comparefn is nil, alphabetical order will be used.

(sortcar 'l_list 'u_comparefn)

RETURNS: a list of the elements of l_list with the *car*'s ordered by the sort function u_comparefn.

SIDE EFFECT: the list l_list is modified rather than copied.

NOTE: Like *sort*, if u_comparefn is nil, alphabetical order will be used.

CHAPTER 3

Arithmetic Functions

This chapter describes FRANZ LISP's functions for doing arithmetic. Often the same function is known by many names. For example, *add* is also *plus*, and *sum*. This is caused by our desire to be compatible with other Lisps. The FRANZ LISP user should avoid using functions with names such as + and * unless their arguments are fixnums. The Lisp compiler takes advantage of these implicit declarations.

An attempt to divide or to generate a floating point result outside of the range of floating point numbers will cause a floating exception signal from the UNIX operating system. The user can catch and process this interrupt if desired (see the description of the *signal* function).

3.1. Simple Arithmetic Functions

(add ['n_arg1 ...])
(plus ['n_arg1 ...])
(sum ['n_arg1 ...])
(+ ['x_arg1 ...])

RETURNS:the sum of the arguments. If no arguments are given, 0 is returned.

NOTE: if the size of the partial sum exceeds the limit of a fixnum, the partial sum will be converted to a bignum. If any of the arguments are flonums, the partial sum will be converted to a flonum when that argument is processed and the result will thus be a flonum. Currently, if in the process of doing the addition a bignum must be converted into a flonum an error message will result.

(add1 'n_arg)
(1+ 'x_arg)

RETURNS:its argument plus 1.

(diff ['n_arg1 ...])
(difference ['n_arg1 ...])
(- ['x_arg1 ...])

RETURNS:the result of subtracting from n_arg1 all subsequent arguments. If no arguments are given, 0 is returned.

NOTE: See the description of add for details on data type conversions and restrictions.

(sub1 'n_arg)
(1- 'x_arg)

RETURNS:its argument minus 1.

(minus 'n_arg)

RETURNS:zero minus n_arg.

(product ['n_arg1 ...])
(times ['n_arg1 ...])
(* ['x_arg1 ...])

RETURNS:the product of all of its arguments. It returns 1 if there are no arguments.

NOTE: See the description of the function *add* for details and restrictions to the automatic data type coercion.

(quotient ['n_arg1 ...])
(/ ['x_arg1 ...])

RETURNS:the result of dividing the first argument by succeeding ones.

NOTE: If there are no arguments, 1 is returned. See the description of the function *add* for details and restrictions of data type coercion. A divide by zero will cause a floating exception interrupt – see the description of the *signal* function.

(*quo 'i_x 'i_y)

RETURNS:the integer part of i_x / i_y.

(Divide 'i_dividend 'i_divisor)

RETURNS:a list whose car is the quotient and whose cadr is the remainder of the division of i_dividend by i_divisor.

NOTE: this is restricted to integer division.

(Emuldiv 'x_fact1 'x_fact2 'x_addn 'x_divisor)

RETURNS:a list of the quotient and remainder of this operation: ((x_fact1 * x_fact2) + (sign extended) x_addn) / x_divisor.

NOTE: this is useful for creating a bignum arithmetic package in Lisp.

3.2. predicates

(numberp 'g_arg**)**

(numbp 'g_arg**)**

RETURNS:t iff g_arg is a number (fixnum, flonum or bignum).

(fixp 'g_arg**)**

RETURNS:t iff g_arg is a fixnum or bignum.

(floatp 'g_arg**)**

RETURNS:t iff g_arg is a flonum.

(evenp 'x_arg**)**

RETURNS:t iff x_arg is even.

(oddp 'x_arg**)**

RETURNS:t iff x_arg is odd.

(zerop 'g_arg**)**

RETURNS:t iff g_arg is a number equal to 0.

(onep 'g_arg**)**

RETURNS:t iff g_arg is a number equal to 1.

(plusp 'n_arg**)**

RETURNS:t iff n_arg is greater than zero.

(minusp 'g_arg**)**

RETURNS:t iff g_arg is a negative number.

(greaterp ['n_arg1 ...])
(> 'fx_arg1 'fx_arg2**)**
(>& 'x_arg1 'x_arg2**)**

RETURNS:t iff the arguments are in a strictly decreasing order.

NOTE: In functions *greaterp* and > the function *difference* is used to compare adjacent values. If any of the arguments are non-numbers, the error message will come from the *difference* function. The arguments to > must be fixnums or both flonums. The arguments to >& must both be fixnums.

(lessp ['n_arg1 ...])
(< 'fx_arg1 'fx_arg2)
(<& 'x_arg1 'x_arg2)

RETURNS:t iff the arguments are in a strictly increasing order.

NOTE: In functions *lessp* and < the function *difference* is used to compare adjacent values. If any of the arguments are non numbers, the error message will come from the *difference* function. The arguments to < may be either fixnums or flonums but must be the same type. The arguments to <& must be fixnums.

(= 'fx_arg1 'fx_arg2)

(=& 'x_arg1 'x_arg2)

RETURNS:t iff the arguments have the same value. The arguments to = must be the either both fixnums or both flonums. The arguments to =& must be fixnums.

3.3. Trignometric Functions

Some of these funtcions are taken from the host math library, and we take no further responsibility for their accuracy.

(cos 'fx_angle)

RETURNS:the (flonum) cosine of fx_angle (which is assumed to be in radians).

(sin 'fx_angle)

RETURNS:the sine of fx_angle (which is assumed to be in radians).

(acos 'fx_arg)

RETURNS:the (flonum) arc cosine of fx_arg in the range 0 to π.

(asin 'fx_arg)

RETURNS:the (flonum) arc sine of fx_arg in the range $-\pi/2$ to $\pi/2$.

(atan 'fx_arg1 'fx_arg2)

RETURNS:the (flonum) arc tangent of fx_arg1/fx_arg2 in the range $-\pi$ to π.

3.4. Bignum/Fixnum Manipulation

(haipart bx_number x_bits**)**

RETURNS: a fixnum (or bignum) which contains the x_bits high bits of *(abs bx_number)* if x_bits is positive, otherwise it returns the *(abs x_bits)* low bits of *(abs bx_number)*.

(haulong bx_number**)**

RETURNS: the number of significant bits in bx_number.

NOTE: the result is equal to the least integer greater to or equal to the base two logarithm of one plus the absolute value of bx_number.

(bignum-leftshift bx_arg x_amount**)**

RETURNS: bx_arg shifted left by x_amount. If x_amount is negative, bx_arg will be shifted right by the magnitude of x_amount.

NOTE: If bx_arg is shifted right, it will be rounded to the nearest even number.

(sticky-bignum-leftshift 'bx_arg 'x_amount**)**

RETURNS: bx_arg shifted left by x_amount. If x_amount is negative, bx_arg will be shifted right by the magnitude of x_amount and rounded.

NOTE: sticky rounding is done this way: after shifting, the low order bit is changed to 1 if any 1's were shifted off to the right.

3.5. Bit Manipulation

(boole 'x_key 'x_v1 'x_v2 ...**)**

RETURNS: the result of the bitwise boolean operation as described in the following table.

NOTE: If there are more than 3 arguments, then evaluation proceeds left to right with each partial result becoming the new value of x_v1. That is,
(boole 'key 'v1 'v2 'v3) ≡ *(boole 'key (boole 'key 'v1 'v2) 'v3)*.
In the following table, * represents bitwise and, + represents bitwise or, ⊕ represents bitwise xor and ¬ represents bitwise negation and is the highest precedence operator.

(boole 'key 'x 'y)								
key	0	1	2	3	4	5	6	7
result	0	x * y	¬ x * y	y	x * ¬ y	x	x ⊕ y	x + y
common names		and			bitclear		xor	or
key	8	9	10	11	12	13	14	15
result	¬ (x + y)	¬(x ⊕ y)	¬ x	¬ x + y	¬ y	x + ¬ y	¬ x + ¬ y	-1
common names	nor	equiv		implies			nand	

(lsh 'x_val 'x_amt)

RETURNS:x_val shifted left by x_amt if x_amt is positive. If x_amt is negative, then *lsh* returns x_val shifted right by the magnitude if x_amt.

NOTE: This always returns a fixnum even for those numbers whose magnitude is so large that they would normally be represented as a bignum, i.e. shifter bits are lost. For more general bit shifters, see *bignum-leftshift* and *sticky-bignum-leftshift.*

(rot 'x_val 'x_amt)

RETURNS:x_val rotated left by x_amt if x_amt is positive. If x_amt is negative, then x_val is rotated right by the magnitude of x_amt.

3.6. Other Functions

As noted above, some of the following functions are inherited from the host math library, with all their virtues and vices.

(abs 'n_arg)
(absval 'n_arg)

RETURNS:the absolute value of n_arg.

(exp 'fx_arg)

RETURNS:*e* raised to the fx_arg power (flonum).

(expt 'n_base 'n_power)

RETURNS:n_base raised to the n_power power.

NOTE: if either of the arguments are flonums, the calculation will be done using *log* and *exp.*

(fact 'x_arg)

RETURNS:x_arg factorial. (fixnum or bignum)

(fix 'n_arg)

RETURNS:a fixnum as close as we can get to n_arg.

NOTE: *fix* will round down. Currently, if n_arg is a flonum larger than the size of a fixnum, this will fail.

(float 'n_arg)

RETURNS:a flonum as close as we can get to n_arg.

NOTE: if n_arg is a bignum larger than the maximum size of a flonum, then a floating exception will occur.

(log 'fx_arg**)**

RETURNS:the natural logarithm of fx_arg.

(max 'n_arg1 ... **)**

RETURNS:the maximum value in the list of arguments.

(min 'n_arg1 ... **)**

RETURNS:the minimum value in the list of arguments.

(mod 'i_dividend 'i_divisor**)**
(remainder 'i_dividend 'i_divisor**)**

RETURNS:the remainder when i_dividend is divided by i_divisor.

NOTE: The sign of the result will have the same sign as i_dividend.

(*mod 'x_dividend 'x_divisor**)**

RETURNS:the balanced representation of x_dividend modulo x_divisor.

NOTE: the range of the balanced representation is abs(x_divisor)/2 to (abs(x_divisor)/2) - x_divisor + 1.

(random ['x_limit]**)**

RETURNS:a fixnum between 0 and x_limit - 1 if x_limit is given. If x_limit is not given, any fixnum, positive or negative, might be returned.

(sqrt 'fx_arg**)**

RETURNS:the square root of fx_arg.

CHAPTER 4

Special Functions

(and [g_arg1 ...])

RETURNS: the value of the last argument if all arguments evaluate to a non-nil value, otherwise *and* returns nil. It returns t if there are no arguments.

NOTE: the arguments are evaluated left to right and evaluation will cease with the first nil encountered.

(apply 'u_func 'l_args)

RETURNS: the result of applying function u_func to the arguments in the list l_args.

NOTE: If u_func is a lambda, then the *(length l_args)* should equal the number of formal parameters for the u_func. If u_func is a nlambda or macro, then l_args is bound to the single formal parameter.

```
; add1 is a lambda of 1 argument
-> (apply 'add1 '(3))
4

; we will define plus1 as a macro which will be equivalent to add1
-> (def plus1 (macro (arg) (list 'add1 (cadr arg))))
plus1
-> (plus1 3)
4

; now if we apply a macro we obtain the form it changes to.
-> (apply 'plus1 '(plus1 3))
(add1 3)

; if we funcall a macro however, the result of the macro is evaled
; before it is returned.
-> (funcall 'plus1 '(plus1 3))
4

; for this particular macro, the car of the arg is not checked
; so that this too will work
-> (apply 'plus1 '(foo 3))
(add1 3)
```

(arg ['x_numb])

RETURNS: if x_numb is specified then the x_numb'*th* argument to the enclosing lexpr If x_numb is not specified then this returns the number of arguments to the enclosing lexpr.

NOTE: it is an error to the interpreter if x_numb is given and out of range.

(break [g_message ['g_pred]])

WHERE: if g_message is not given it is assumed to be the null string, and if g_pred is not given it is assumed to be t.

RETURNS: the value of *(*break 'g_pred 'g_message)*

(*break 'g_pred 'g_message)

RETURNS: nil immediately if g_pred is nil, else the value of the next (return 'value) expression typed in at top level.

SIDE EFFECT: If the predicate, g_pred, evaluates to non-null, the lisp system stops and prints out 'Break ' followed by g_message. It then enters a break loop which allows one to interactively debug a program. To continue execution from a break you can use the *return* function. to return to top level or another break level, you can use *retbrk* or *reset*.

(caseq 'g_key-form l_clause1 ...)

WHERE: l_clause*i* is a list of the form (g_comparator ['g_form*i* ...]). The comparators may be symbols, small fixnums, a list of small fixnums or symbols.

NOTE: The way caseq works is that it evaluates g_key-form, yielding a value we will call the selector. Each clause is examined until the selector is found consistent with the comparator. For a symbol, or a fixnum, this means the two must be *eq*. For a list, this means that the selector must be *eq* to some element of the list.

The comparator consisting of the symbol t has special semantics: it matches anything, and consequently, should be the last comparator.

In any case, having chosen a clause, *caseq* evaluates each form within that clause and

RETURNS: the value of the last form. If no comparators are matched, *caseq* returns nil.

Here are two ways of defining the same function:

```
->(defun fate (personna)
        (caseq personna
          (cow '(jumped over the moon))
          (cat '(played nero))
          ((dish spoon) '(ran away with each other))
          (t '(lived happily ever after))))
fate
->(defun fate (personna)
        (cond
                ((eq personna 'cow) '(jumped over the moon))
                ((eq personna 'cat) '(played nero))
                ((memq personna '(dish spoon)) '(ran away with each other))
                (t '(lived happily ever after))))
fate
```

(catch g_exp [ls_tag]**)**

WHERE: if ls_tag is not given, it is assumed to be nil.

RETURNS: the result of *(*catch 'ls_tag g_exp)*

NOTE: catch is defined as a macro.

(*catch 'ls_tag g_exp**)**

WHERE: ls_tag is either a symbol or a list of symbols.

RETURNS: the result of evaluating g_exp or the value thrown during the evaluation of g_exp.

SIDE EFFECT: this first sets up a 'catch frame' on the lisp runtime stack. Then it begins to evaluate g_exp. If g_exp evaluates normally, its value is returned. If, however, a value is thrown during the evaluation of g_exp then this *catch will return with that value if one of these cases is true:

(1) the tag thrown to is ls_tag

(2) ls_tag is a list and the tag thrown to is a member of this list

(3) ls_tag is nil.

NOTE: Errors are implemented as a special kind of throw. A catch with no tag will not catch an error but a catch whose tag is the error type will catch that type of error. See Chapter 10 for more information.

(comment [g_arg ...])

RETURNS:the symbol comment.

NOTE: This does absolutely nothing.

(cond [l_clause1 ...])

RETURNS:the last value evaluated in the first clause satisfied. If no clauses are satisfied then nil is returned.

NOTE: This is the basic conditional 'statement' in lisp. The clauses are processed from left to right. The first element of a clause is evaluated. If it evaluated to a non-null value then that clause is satisfied and all following elements of that clause are evaluated. The last value computed is returned as the value of the cond. If there is just one element in the clause then its value is returned. If the first element of a clause evaluates to nil, then the other elements of that clause are not evaluated and the system moves to the next clause.

(cvttointlisp)

SIDE EFFECT: The reader is modified to conform with the Interlisp syntax. The character % is made the escape character and special meanings for comma, backquote and backslash are removed. Also the reader is told to convert upper case to lower case.

(cvttofranzlisp)

SIDE EFFECT: FRANZ LISP's default syntax is reinstated. One would run this function after having run any of the other *cvtto-* functions. Backslash is made the escape character, super-brackets work again, and the reader distinguishes between upper and lower case.

(cvttomaclisp)

SIDE EFFECT: The reader is modified to conform with Maclisp syntax. The character / is made the escape character and the special meanings for backslash, left and right bracket are removed. The reader is made case-insensitive.

(cvttoucilisp)

SIDE EFFECT: The reader is modified to conform with UCI Lisp syntax. The character / is made the escape character, tilde is made the comment character, exclamation point takes on the unquote function normally held by comma, and backslash, comma, semicolon become normal characters. Here too, the reader is made case-insensitive.

(debug s_msg**)**

SIDE EFFECT: Enter the Fixit package described in Chapter 15. This package allows you to examine the evaluation stack in detail. To leave the Fixit package type 'ok'.

(debugging 'g_arg**)**

SIDE EFFECT: If g_arg is non-null, Franz unlinks the transfer tables, does a *(*rset t)* to turn on evaluation monitoring and sets the all-error catcher (ER%all) to be *debug-err-handler*. If g_arg is nil, all of the above changes are undone.

(declare [g_arg ...]**)**

RETURNS:nil

NOTE: this is a no-op to the evaluator. It has special meaning to the compiler (see Chapter 12).

(def s_name (s_type l_argl g_exp1 ...))

WHERE: s_type is one of lambda, nlambda, macro or lexpr.

RETURNS:s_name

SIDE EFFECT: This defines the function s_name to the lisp system. If s_type is nlambda or macro then the argument list l_argl must contain exactly one non-nil symbol.

(defmacro s_name l_arg g_exp1 ...)
(defcmacro s_name l_arg g_exp1 ...)

RETURNS:s_name

SIDE EFFECT: This defines the macro s_name. *defmacro* makes it easy to write macros since it makes the syntax just like *defun*. Further information on *defmacro* is in §8.3.2. *defcmacro* defines compiler-only macros, or cmacros. A cmacro is stored on the property list of a symbol under the indicator **cmacro**. Thus a function can have a normal definition and a cmacro definition. For an example of the use of cmacros, see the definitions of nthcdr and nth in /usr/lib/lisp/common2.l

(defun s_name [s_mtype] ls_argl g_exp1 ...)

WHERE: s_mtype is one of fexpr, expr, args or macro.

RETURNS:s_name

SIDE EFFECT: This defines the function s_name.

NOTE: this exists for Maclisp compatibility, it is just a macro which changes the defun form to the def form. An s_mtype of fexpr is converted to nlambda and of expr to lambda. Macro remains the same. If ls_arg1 is a non-nil symbol, then the type is assumed to be lexpr and ls_arg1 is the symbol which is bound to the number of args when the function is entered.
For compatibility with the Lisp Machine Lisp, there are three types of optional parameters that can occur in ls_argl: *&optional* declares that the following symbols are optional, and may or may not appear in the argument list to the function, *&rest symbol* declares that all forms in the function call that are not accounted for by previous lambda bindings are to be assigned to *symbol*, and *&aux form1 ... formn* declares that the *formi* are either symbols, in which case they are lambda

bound to **nil**, or lists, in which case the first element of the list is lambda bound to the second, evaluated element.

```
; def and defun here are used to define identical functions
; you can decide for yourself which is easier to use.
-> (def append1 (lambda (lis extra) (append lis (list extra))))
append1

-> (defun append1 (lis extra) (append lis (list extra)))
append1

; Using the & forms...
-> (defun test (a b &optional c &aux (retval 0) &rest z)
        (if c then (msg "Optional arg present" N
                        "c is " c N))
        (msg "rest is " z N
             "retval is " retval N))
test
-> (test 1 2 3 4)
Optional arg present
c is 3
rest is (4)
retval is 0
```

(**defvar** s_variable ['g_init])

RETURNS:s_variable.

NOTE: This form is put at the top level in files, like *defun*.

SIDE EFFECT: This declares s_variable to be special. If g_init is present and s_variable is unbound when the file is read in, s_variable will be set to the value of g_init. An advantage of '(defvar foo)' over '(declare (special foo))' is that if a file containing defvars is loaded (or fasl'ed) in during compilation, the variables mentioned in the defvar's will be declared special. The only way to have that effect with '(declare (special foo))' is to *include* the file.

(**do** l_vrbs l_test g_exp1 ...)

RETURNS:the last form in the cdr of l_test evaluated, or a value explicitly given by a return evaluated within the do body.

NOTE: This is the basic iteration form for FRANZ LISP. l_vrbs is a list of zero or more var-init-repeat forms. A var-init-repeat form looks like:

(s_name [g_init [g_repeat]])

There are three cases depending on what is present in the form. If just s_name is present, this means that when the do is entered, s_name is lambda-bound to nil and is never modified by the system (though the program is certainly free to modify its value). If the form is (s_name 'g_init) then the only difference is that s_name is lambda-bound to the value of g_init instead of nil. If g_repeat is also present then s_name is lambda-bound to g_init when the loop is entered and after each pass through the do body s_name is bound to the value of g_repeat.

l_test is either nil or has the form of a cond clause. If it is nil then the do body will be evaluated only once and the do will return nil. Otherwise, before the do body is evaluated the car of l_test is evaluated and if the result is non-null, this

signals an end to the looping. Then the rest of the forms in l_test are evaluated and the value of the last one is returned as the value of the do. If the cdr of l_test is nil, then nil is returned -- thus this is not exactly like a cond clause.
g_exp1 and those forms which follow constitute the do body. A do body is like a prog body and thus may have labels and one may use the functions go and return. The sequence of evaluations is this:

(1) the init forms are evaluated left to right and stored in temporary locations.

(2) Simultaneously all do variables are lambda bound to the value of their init forms or nil.

(3) If l_test is non-null, then the car is evaluated and if it is non-null, the rest of the forms in l_test are evaluated and the last value is returned as the value of the do.

(4) The forms in the do body are evaluated left to right.

(5) If l_test is nil the do function returns with the value nil.

(6) The repeat forms are evaluated and saved in temporary locations.

(7) The variables with repeat forms are simultaneously bound to the values of those forms.

(8) Go to step 3.

NOTE: there is an alternate form of do which can be used when there is only one do variable. It is described next.

```
; this is  a simple function which numbers the elements of a list.
; It uses a do function with two local variables.
-> (defun printem (lis)
            (do ((xx lis (cdr xx))
                 (i 1 (1+ i)))
                ((null xx) (patom "all done") (terpr))
                (print i)
                (patom ": ")
                (print (car xx))
                (terpr)))
printem
-> (printem '(a b c d))
1: a
2: b
3: c
4: d
all done
nil
->
```

(do s_name g_init g_repeat g_test g_exp1 ...**)**

NOTE: this is another, less general, form of do. It is evaluated by:

(1) evaluating g_init

(2) lambda binding s_name to value of g_init

(3) g_test is evaluated and if it is not nil the do function returns with nil.

(4) the do body is evaluated beginning at g_exp1.

(5) the repeat form is evaluated and stored in s_name.

(6) go to step 3.

RETURNS: nil

(environment [l_when1 l_what1 l_when2 l_what2 ...]**)**
(environment-maclisp [l_when1 l_what1 l_when2 l_what2 ...]**)**
(environment-lmlisp [l_when1 l_what1 l_when2 l_what2 ...]**)**

WHERE: the when's are a subset of (eval compile load), and the symbols have the same meaning as they do in 'eval-when'.

The what's may be
(files file1 file2 ... fileN),
which insure that the named files are loaded. To see if file*i* is loaded, it looks for a 'version' property under file*i*'s property list. Thus to prevent multiple loading, you should put
(putprop 'myfile t 'version),
at the end of myfile.l.

Another acceptable form for a what is
(syntax type)
Where type is either maclisp, intlisp, ucilisp, franzlisp.

SIDE EFFECT: *environment-maclisp* sets the environment to that which 'liszt -m' would generate.

environment-lmlisp sets up the lisp machine environment. This is like maclisp but it has additional macros.

For these specialized environments, only the **files** clauses are useful.
(environment-maclisp (compile eval) (files foo bar))

RETURNS: the last list of files requested.

(err ['s_value [nil]]**)**

RETURNS: nothing (it never returns).

SIDE EFFECT: This causes an error and if this error is caught by an *errset* then that *errset* will return s_value instead of nil. If the second arg is given, then it must be nil (MACLisp compatibility).

(error ['s_message1 ['s_message2]])

RETURNS: nothing (it never returns).

SIDE EFFECT: s_message1 and s_message2 are *patom*ed if they are given and then *err* is called (with no arguments), which causes an error.

(errset g_expr [s_flag])

RETURNS: a list of one element, which is the value resulting from evaluating g_expr. If an error occurs during the evaluation of g_expr, then the locus of control will return to the *errset* which will then return nil (unless the error was caused by a call to *err*, with a non-null argument).

SIDE EFFECT: S_flag is evaluated before g_expr is evaluated. If s_flag is not given, then it is assumed to be t. If an error occurs during the evaluation of g_expr, and s_flag evaluated to a non-null value, then the error message associated with the error is printed before control returns to the errset.

(eval 'g_val ['x_bind-pointer])

RETURNS: the result of evaluating g_val.

NOTE: The evaluator evaluates g_val in this way:

If g_val is a symbol, then the evaluator returns its value. If g_val had never been assigned a value, then this causes an 'Unbound Variable' error. If x_bind-pointer is given, then the variable is evaluated with respect to that pointer (see *evalframe* for details on bind-pointers).

If g_val is of type value, then its value is returned. If g_val is of any other type than list, g_val is returned.

If g_val is a list object then g_val is either a function call or array reference. Let g_car be the first element of g_val. We continually evaluate g_car until we end up with a symbol with a non-null function binding or a non-symbol. Call what we end up with: g_func.

G_func must be one of three types: list, binary or array. If it is a list then the first element of the list, which we shall call g_functype, must be either lambda, nlambda, macro or lexpr. If g_func is a binary, then its discipline, which we shall call g_functype, is either lambda, nlambda, macro or a string. If g_func is an array then this form is evaluated specially, see Chapter 9 on arrays. If g_func is a list or binary, then g_functype will determine how the arguments to this function, the cdr of g_val, are processed. If g_functype is a string, then this is a foreign function call (see §8.5 for more details).

If g_functype is lambda or lexpr, the arguments are evaluated (by calling *eval* recursively) and stacked. If g_functype is nlambda then the argument list is stacked. If g_functype is macro then the entire form, g_val is stacked.

Next, the formal variables are lambda bound. The formal variables are the cadr of g_func. If g_functype is nlambda, lexpr or macro, there should only be one formal variable. The values on the stack are lambda bound to the formal variables except in the case of a lexpr, where the number of actual arguments is bound to the formal variable.

After the binding is done, the function is invoked, either by jumping to the entry

point in the case of a binary or by evaluating the list of forms beginning at cddr g_func. The result of this function invocation is returned as the value of the call to eval.

(evalframe 'x_pdlpointer)

RETURNS: an evalframe descriptor for the evaluation frame just before x_pdlpointer. If x_pdlpointer is nil, it returns the evaluation frame of the frame just before the current call to *evalframe.*

NOTE: An evalframe descriptor describes a call to *eval, apply* or *funcall.* The form of the descriptor is
(type pdl-pointer expression bind-pointer np-index lbot-index)
where type is 'eval' if this describes a call to *eval* or 'apply' if this is a call to *apply* or *funcall.* pdl-pointer is a number which describes this context. It can be passed to *evalframe* to obtain the next descriptor and can be passed to *freturn* to cause a return from this context. bind-pointer is the size of variable binding stack when this evaluation began. The bind-pointer can be given as a second argument to *eval* to order to evaluate variables in the same context as this evaluation. If type is 'eval' then expression will have the form *(function-name arg1 ...).* If type is 'apply' then expression will have the form *(function-name (arg1 ...)).* np-index and lbot-index are pointers into the argument stack (also known as the *namestack* array) at the time of call. lbot-index points to the first argument, np-index points one beyond the last argument.
In order for there to be enough information for *evalframe* to return, you must call *(*rset t).*

EXAMPLE: *(progn (evalframe nil))*
returns *(eval 2147478600 (progn (evalframe nil)) 1 8 7)*

(evalhook 'g_form 'su_evalfunc ['su_funcallfunc])

RETURNS: the result of evaluating g_form after lambda binding 'evalhook' to su_evalfunc and, if it is given, lambda binding 'funcallhook' to su_funcallhook.

NOTE: As explained in §14.4, the function *eval* may pass the job of evaluating a form to a user 'hook' function when various switches are set. The hook function normally prints the form to be evaluated on the terminal and then evaluates it by calling *evalhook. Evalhook* does the lambda binding mentioned above and then calls *eval* to evaluate the form after setting an internal switch to tell *eval* not to call the user's hook function just this one time. This allows the evaluation process to advance one step and yet insure that further calls to *eval* will cause traps to the hook function (if su_evalfunc is non-null).
In order for *evalhook* to work, *(*rset t)* and *(sstatus evalhook t)* must have been done previously.

(exec s_arg1 ...**)**

RETURNS: the result of forking and executing the command named by concatenating the s_arg*i* together with spaces in between.

(exece 's_fname ['l_args ['l_envir]]**)**

RETURNS: the error code from the system if it was unable to execute the command s_fname with arguments l_args and with the environment set up as specified in l_envir. If this function is successful, it will not return, instead the lisp system will be overlaid by the new command.

(freturn 'x_pdl-pointer 'g_retval**)**

RETURNS: g_retval from the context given by x_pdl-pointer.

NOTE: A pdl-pointer denotes a certain expression currently being evaluated. The pdl-pointer for a given expression can be obtained from *evalframe*.

(frexp 'f_arg**)**

RETURNS: a list cell *(exponent . mantissa)* which represents the given flonum

NOTE: The exponent will be a fixnum, the mantissa a 56 bit bignum. If you think of the the binary point occurring right after the high order bit of mantissa, then f_arg = 2^{exponent} * mantissa.

(funcall 'u_func ['g_arg1 ...]**)**

RETURNS: the value of applying function u_func to the arguments g_arg*i* and then evaluating that result if u_func is a macro.

NOTE: If u_func is a macro or nlambda then there should be only one g_arg. *funcall* is the function which the evaluator uses to evaluate lists. If *foo* is a lambda or lexpr or array, then *(funcall 'foo 'a 'b 'c)* is equivalent to *(foo 'a 'b 'c)*. If *foo* is a nlambda then *(funcall 'foo '(a b c))* is equivalent to *(foo a b c)*. Finally, if *foo* is a macro then *(funcall 'foo '(foo a b c))* is equivalent to *(foo a b c)*.

(funcallhook 'l_form 'su_funcallfunc ['su_evalfunc]**)**

RETURNS: the result of *funcall*ing the *(car l_form)* on the already evaluated arguments in the *(cdr l_form)* after lambda binding 'funcallhook' to su_funcallfunc and, if it is given, lambda binding 'evalhook' to su_evalhook.

NOTE: This function is designed to continue the evaluation process with as little work as possible after a funcallhook trap has occurred. It is for this reason that the form of l_form is unorthodox: its *car* is the name of the function to call and its *cdr* are a list of arguments to stack (without evaluating again) before calling the given function. After stacking the arguments but before calling *funcall* an internal switch is set to prevent *funcall* from passing the job of funcalling to su_funcallfunc. If *funcall* is called recursively in funcalling l_form and if su_funcallfunc is non-null, then the arguments to *funcall* will actually be given to su_funcallfunc (a lexpr) to be funcalled.
In order for *evalhook* to work, *(*rset t)* and *(sstatus evalhook t)* must have been done previously. A more detailed description of *evalhook* and *funcallhook* is given in Chapter 14.

(function u_func)

RETURNS: the function binding of u_func if it is an symbol with a function binding otherwise u_func is returned.

(getdisc 'y_func)

RETURNS: the discipline of the machine coded function (either lambda, nlambda or macro).

(go g_labexp)

WHERE: g_labexp is either a symbol or an expression.

SIDE EFFECT: If g_labexp is an expression, that expression is evaluated and should result in a symbol. The locus of control moves to just following the symbol g_labexp in the current prog or do body.

NOTE: this is only valid in the context of a prog or do body. The interpreter and compiler will allow non-local *go*'s although the compiler won't allow a *go* to leave a function body. The compiler will not allow g_labexp to be an expression.

(if 'g_a 'g_b)
(if 'g_a 'g_b 'g_c ...)
(if 'g_a then 'g_b [...] [elseif 'g_c then 'g_d ...] [else 'g_e [...])
(if 'g_a then 'g_b [...] [elseif 'g_c thenret] [else 'g_d [...])

NOTE: The various forms of *if* are intended to be a more readable conditional statement, to be used in place of *cond*. There are two varieties of *if*, with keywords, and without. The keyword-less variety is inherited from common Maclisp usage. A keyword-less, two argument *if* is equivalent to a one-clause *cond*, i.e. (*cond* (a b)). Any other keyword-less *if* must have at least three arguments. The first two arguments are the first clause of the equivalent *cond*, and all remaining arguments are shoved into a second clause beginning with t. Thus, the second form of *if* is equivalent to

(*cond* (a b) (t c ...)).

The keyword variety has the following grouping of arguments: a predicate, a then-clause, and optional else-clause. The predicate is evaluated, and if the result is non-nil, the then-clause will be performed, in the sense described below. Otherwise, (i.e. the result of the predicate evaluation was precisely nil), the else-clause will be performed.

Then-clauses will either consist entirely of the single keyword **thenret**, or will start with the keyword **then**, and be followed by at least one general expression. (These general expressions must not be one of the keywords.) To actuate a **thenret** means to cease further evaluation of the *if*, and to return the value of the predicate just calculated. The performance of the longer clause means to evaluate each general expression in turn, and then return the last value calculated.

The else-clause may begin with the keyword **else** and be followed by at least one general expression. The rendition of this clause is just like that of a then-clause. An else-clause may begin alternatively with the keyword **elseif**, and be followed (recursively) by a predicate, then-clause, and optional else-clause. Evaluation of this clause, is just evaluation of an *if*-form, with the same predicate, then- and else-clauses.

(I-throw-err 'l_token**)**

WHERE: l_token is the *cdr* of the value returned from a **catch* with the tag ER%unwind-protect.

RETURNS: nothing (never returns in the current context)

SIDE EFFECT: The error or throw denoted by l_token is continued.

NOTE: This function is used to implement *unwind-protect* which allows the processing of a transfer of control though a certain context to be interrupted, a user function to be executed and than the transfer of control to continue. The form of l_token is either
(t tag value) for a throw or
(nil type message valret contuab uniqueid [arg ...]) for an error.
This function is not to be used for implementing throws or errors and is only documented here for completeness.

(let l_args g_exp1 ... g_exprn**)**

RETURNS: the result of evaluating g_exprn within the bindings given by l_args.

NOTE: l_args is either nil (in which case *let* is just like *progn*) or it is a list of binding objects. A binding object is a list *(symbol expression)*. When a *let* is entered, all of the expressions are evaluated and then simultaneously lambda-bound to the corresponding symbols. In effect, a *let* expression is just like a lambda expression except the symbols and their initial values are next to each other, making the expression easier to understand. There are some added features to the *let* expression: A binding object can just be a symbol, in which case the expression corresponding to that symbol is 'nil'. If a binding object is a list and the first element of that list is another list, then that list is assumed to be a binding template and *let* will do a *desetq* on it.

(let* l_args g_exp1 ... g_expn**)**

RETURNS: the result of evaluating g_exprn within the bindings given by l_args.

NOTE: This is identical to *let* except the expressions in the binding list l_args are evaluated and bound sequentially instead of in parallel.

(lexpr-funcall 'g_function ['g_arg1 ...] 'l_argn**)**

NOTE: This is a cross between funcall and apply. The last argument, must be a list (possibly empty). The element of list arg are stack and then the function is funcalled.

EXAMPLE: (lexpr-funcall 'list 'a '(b c d)) is the same as
(funcall 'list 'a 'b 'c 'd)

(listify 'x_count**)**

RETURNS: a list of x_count of the arguments to the current function (which must be a lexpr).

NOTE: normally arguments 1 through x_count are returned. If x_count is negative then a list of last abs(x_count) arguments are returned.

(map 'u_func 'l_arg1 ...)

RETURNS:l_arg1

NOTE: The function u_func is applied to successive sublists of the l_arg*i*. All sublists should have the same length.

(mapc 'u_func 'l_arg1 ...)

RETURNS:l_arg1.

NOTE: The function u_func is applied to successive elements of the argument lists. All of the lists should have the same length.

(mapcan 'u_func 'l_arg1 ...)

RETURNS:nconc applied to the results of the functional evaluations.

NOTE: The function u_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcar 'u_func 'l_arg1 ...)

RETURNS:a list of the values returned from the functional application.

NOTE: the function u_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcon 'u_func 'l_arg1 ...)

RETURNS:nconc applied to the results of the functional evaluation.

NOTE: the function u_func is applied to successive sublists of the argument lists. All sublists should have the same length.

(maplist 'u_func 'l_arg1 ...)

RETURNS:a list of the results of the functional evaluations.

NOTE: the function u_func is applied to successive sublists of the arguments lists. All sublists should have the same length.

Readers may find the following summary table useful in remembering the differences between the six mapping functions:

	Value returned is		
Argument to functional is	l_arg1	list of results	*nconc* of results
elements of list	mapc	mapcar	mapcan
sublists	map	maplist	mapcon

(mfunction t_entry 's_disc**)**

RETURNS:a lisp object of type binary composed of t_entry and s_disc.

NOTE: t_entry is a pointer to the machine code for a function, and s_disc is the discipline (e.g. lambda).

(oblist)

RETURNS:a list of all symbols on the oblist.

(or [g_arg1 ...]**)**

RETURNS:the value of the first non-null argument or nil if all arguments evaluate to nil.

NOTE: Evaluation proceeds left to right and stops as soon as one of the arguments evaluates to a non-null value.

(prog l_vrbls g_exp1 ...**)**

RETURNS:the value explicitly given in a return form or else nil if no return is done by the time the last g_exp*i* is evaluated.

NOTE: the local variables are lambda-bound to nil, then the g_exp*i* are evaluated from left to right. This is a prog body (obviously) and this means than any symbols seen are not evaluated, but are treated as labels. This also means that return's and go's are allowed.

(prog1 'g_exp1 ['g_exp2 ...]**)**

RETURNS:g_exp1

(prog2 'g_exp1 'g_exp2 ['g_exp3 ...]**)**

RETURNS:g_exp2

NOTE: the forms are evaluated from left to right and the value of g_exp2 is returned.

(progn 'g_exp1 ['g_exp2 ...]**)**

RETURNS:the last g_exp*i*.

(progv 'l_locv 'l_initv g_exp1 ...**)**

WHERE: l_locv is a list of symbols and l_initv is a list of expressions.

RETURNS:the value of the last g_exp*i* evaluated.

NOTE: The expressions in l_initv are evaluated from left to right and then lambda-bound to the symbols in l_locv. If there are too few expressions in l_initv then the missing values are assumed to be nil. If there are too many expressions in l_initv then the extra ones are ignored (although they are evaluated). Then the g_exp*i* are evaluated left to right. The body of a progv is like the body of a progn, it is *not* a prog body. (C.f. *let*)

(purcopy 'g_exp**)**

RETURNS:a copy of g_exp with new pure cells allocated wherever possible.

NOTE: pure space is never swept up by the garbage collector, so this should only be done on expressions which are not likely to become garbage in the future. In certain cases, data objects in pure space become read-only after a *dumplisp* and then an attempt to modify the object will result in an illegal memory reference.

(purep 'g_exp**)**

RETURNS:t iff the object g_exp is in pure space.

(putd 's_name 'u_func**)**

RETURNS:u_func

SIDE EFFECT: this sets the function binding of symbol s_name to u_func.

(return ['g_val]**)**

RETURNS:g_val (or nil if g_val is not present) from the enclosing prog or do body.

NOTE: this form is only valid in the context of a prog or do body.

(selectq 'g_key-form [l_clause1 ...]**)**

NOTE: This function is just like *caseq* (see above), except that the symbol **otherwise** has the same semantics as the symbol t, when used as a comparator.

(setarg 'x_argnum 'g_val**)**

WHERE: x_argnum is greater than zero and less than or equal to the number of arguments to the lexpr.

RETURNS:g_val

SIDE EFFECT: the lexpr's x_argnum'th argument is set to g-val.

NOTE: this can only be used within the body of a lexpr.

(throw 'g_val [s_tag]**)**

WHERE: if s_tag is not given, it is assumed to be nil.

RETURNS:the value of *(*throw 's_tag 'g_val)*.

(*throw 's_tag 'g_val**)**

RETURNS:g_val from the first enclosing catch with the tag s_tag or with no tag at all.

NOTE: this is used in conjunction with **catch* to cause a clean jump to an enclosing context.

(unwind-protect g_protected [g_cleanup1 ...])

RETURNS: the result of evaluating g_protected.

NOTE: Normally g_protected is evaluated and its value remembered, then the g_cleanup*i* are evaluated and finally the saved value of g_protected is returned. If something should happen when evaluating g_protected which causes control to pass through g_protected and thus through the call to the unwind-protect, then the g_cleanup*i* will still be evaluated. This is useful if g_protected does something sensitive which must be cleaned up whether or not g_protected completes.

CHAPTER 5

Input/Output

The following functions are used to read from and write to external devices (e.g. files) and programs (through pipes). All I/O goes through the lisp data type called the port. A port may be open for either reading or writing, but usually not both simultaneously (see *fileopen*). There are only a limited number of ports (20) and they will not be reclaimed unless they are *closed*. All ports are reclaimed by a *resetio* call, but this drastic step won't be necessary if the program closes what it uses.

If a port argument is not supplied to a function which requires one, or if a bad port argument (such as nil) is given, then FRANZ LISP will use the default port according to this scheme: If input is being done then the default port is the value of the symbol **piport** and if output is being done then the default port is the value of the symbol **poport**. Furthermore, if the value of piport or poport is not a valid port, then the standard input or standard output will be used, respectively.

The standard input and standard output are usually the keyboard and terminal display unless your job is running in the background and its input or output is connected to a pipe. All output which goes to the standard output will also go to the port **ptport** if it is a valid port. Output destined for the standard output will not reach the standard output if the symbol **^w** is non nil (although it will still go to **ptport** if **ptport** is a valid port).

Some of the functions listed below reference files directly. FRANZ LISP has borrowed a convenient shorthand notation from */bin/csh*, concerning naming files. If a file name begins with ~ (tilde), and the symbol **tilde-expansion**

is bound to something other than nil, then FRANZ LISP expands the file name. It takes the string of characters between the leading tilde, and the first slash as a user-name. Then, that initial segment of the filename is replaced by the home directory of the user. The null username is taken to be the current user.

FRANZ LISP keeps a cache of user home directory information, to minimize searching the password file. Tilde-expansion is performed in the following functions: *cfasl, chdir, fasl, ffasl, fileopen, infile, load, outfile, probef, sys:access, sys:unlink.*

(cfasl 'st_file 'st_entry 'st_funcname ['st_disc ['st_library]])

RETURNS:t

SIDE EFFECT: This is used to load in a foreign function (see §8.4). The object file st_file is loaded into the lisp system. St_entry should be an entry point in the file just loaded. The function binding of the symbol s_funcname will be set to point to st_entry, so that when the lisp function s_funcname is called, st_entry will be run. st_disc is the discipline to be given to s_funcname. st_disc defaults to "subroutine" if it is not given or if it is given as nil. If st_library is non-null, then after st_file is loaded, the libraries given in st_library will be searched to resolve external references. The form of st_library should be something like "-lm". The C library (" -lc ") is always searched so when loading in a C file you probably won't

need to specify a library. For Fortran files, you should specify "-lF77" and if you are doing any I/O, the library entry should be "-lI77 -lF77". For Pascal files "-lpc" is required.

NOTE: This function may be used to load the output of the assembler, C compiler, Fortran compiler, and Pascal compiler but NOT the lisp compiler (use *fasl* for that). If a file has more than one entry point, then use *getaddress* to locate and setup other foreign functions.
It is an error to load in a file which has a global entry point of the same name as a global entry point in the running lisp. As soon as you load in a file with *cfasl*, its global entry points become part of the lisp's entry points. Thus you cannot *cfasl* in the same file twice unless you use *removeaddress* to change certain global entry points to local entry points.

(close 'p_port**)**

RETURNS:t

SIDE EFFECT: the specified port is drained and closed, releasing the port.

NOTE: The standard defaults are not used in this case since you probably never want to close the standard output or standard input.

(cprintf 'st_format 'xfst_val ['p_port]**)**

RETURNS:xfst_val

SIDE EFFECT: The UNIX formatted output function printf is called with arguments st_format and xfst_val. If xfst_val is a symbol then its print name is passed to printf. The format string may contain characters which are just printed literally and it may contain special formatting commands preceded by a percent sign. The complete set of formatting characters is described in the UNIX manual. Some useful ones are %d for printing a fixnum in decimal, %f or %e for printing a flonum, and %s for printing a character string (or print name of a symbol).

EXAMPLE:*(cprintf "Pi equals %f" 3.14159)* prints 'Pi equals 3.14159'

(drain ['p_port]**)**

RETURNS:nil

SIDE EFFECT: If this is an output port then the characters in the output buffer are all sent to the device. If this is an input port then all pending characters are flushed. The default port for this function is the default output port.

(ex [s_filename]**)**
(vi [s_filename]**)**
(exl [s_filename]**)**
(vil [s_filename]**)**

RETURNS:nil

SIDE EFFECT: The lisp system starts up an editor on the file named as the argument. It will try appending .l to the file if it can't find it. The functions *exl* and *vil* will load the file after you finish editing it. These functions will also remember the name of the file so that on subsequent invocations, you don't need to provide the argument.

NOTE: These functions do not evaluate their argument.

(fasl 'st_name ['st_mapf ['g_warn]])

WHERE: st_mapf and g_warn default to nil.

RETURNS:t if the function succeeded, nil otherwise.

SIDE EFFECT: this function is designed to load in an object file generated by the lisp compiler Liszt. File names for object files usually end in '.o', so *fasl* will append '.o' to st_name (if it is not already present). If st_mapf is non nil, then it is the name of the map file to create. *Fasl* writes in the map file the names and addresses of the functions it loads and defines. Normally the map file is created (i.e. truncated if it exists), but if *(sstatus appendmap t)* is done then the map file will be appended. If g_warn is non nil and if a function is loaded from the file which is already defined, then a warning message will be printed.

NOTE: *fasl* only looks in the current directory for the file to load. The function *load* looks through a user-supplied search path and will call *fasl* if it finds a file with the same root name and a '.o' extension. In most cases the user would be better off using the function *load* rather than calling *fasl* directly.

(ffasl 'st_file 'st_entry 'st_funcname ['st_discipline ['st_library]])

RETURNS:the binary object created.

SIDE EFFECT: the Fortran object file st_file is loaded into the lisp system. St_entry should be an entry point in the file just loaded. A binary object will be created and its entry field will be set to point to st_entry. The discipline field of the binary will be set to st_discipline or "subroutine" by default. If st_library is present and non-null, then after st_file is loaded, the libraries given in st_library will be searched to resolve external references. The form of st_library should be something like "-lS -ltermcap". In any case, the standard Fortran libraries will be searched also to resolve external references.

NOTE: in F77 on Unix, the entry point for the fortran function foo is named '_foo_'.

(filepos 'p_port ['x_pos])

RETURNS:the current position in the file if x_pos is not given or else x_pos if x_pos is given.

SIDE EFFECT: If x_pos is given, the next byte to be read or written to the port will be at position x_pos.

(filestat 'st_filename)

RETURNS:a vector containing various numbers which the UNIX operating system assigns to files. if the file doesn't exist, an error is invoked. Use *probef* to determine if the file exists.

NOTE: The individual entries can be accesed by mnemonic functions of the form filestat:*field*, where field may be any of atime, ctime, dev, gid, ino, mode,mtime, nlink, rdev, size, type, uid. See the UNIX programmers manual for a more detailed description of these quantities.

(flatc 'g_form ['x_max])

RETURNS: the number of characters required to print g_form using *patom*. If x_max is given and if *flatc* determines that it will return a value greater than x_max, then it gives up and returns the current value it has computed. This is useful if you just want to see if an expression is larger than a certain size.

(flatsize 'g_form ['x_max])

RETURNS: the number of characters required to print g_form using *print*. The meaning of x_max is the same as for flatc.

NOTE: Currently this just *explode*'s g_form and checks its length.

(fileopen 'st_filename 'st_mode)

RETURNS: a port for reading or writing (depending on st_mode) the file st_name.

SIDE EFFECT: the given file is opened (or created if opened for writing and it doesn't yet exist).

NOTE: this function call provides a direct interface to the operating system's fopen function. The mode may be more than just "r" for read, "w" for write or "a" for append. The modes "r+", "w+" and "a+" permit both reading and writing on a port provided that *fseek* is done between changes in direction. See the UNIX manual description of fopen for more details. This routine does not look through a search path for a given file.

(fseek 'p_port 'x_offset 'x_flag)

RETURNS: the position in the file after the function is performed.

SIDE EFFECT: this function positions the read/write pointer before a certain byte in the file. If x_flag is 0 then the pointer is set to x_offset bytes from the beginning of the file. If x_flag is 1 then the pointer is set to x_offset bytes from the current location in the file. If x_flag is 2 then the pointer is set to x_offset bytes from the end of the file.

(infile 's_filename)

RETURNS: a port ready to read s_filename.

SIDE EFFECT: this tries to open s_filename and if it cannot or if there are no ports available it gives an error message.

NOTE: to allow your program to continue on a file-not-found error, you can use something like:

```
(cond ((null (setq myport (car (errset (infile name) nil))))
       (patom "couldn't open the file")))
```

which will set myport to the port to read from if the file exists or will print a message if it couldn't open it and also set myport to nil. To simply determine if a file exists, use *probef*.

(load 's_filename ['st_map ['g_warn]])

RETURNS:t

NOTE: The function of *load* has changed since previous releases of FRANZ LISP and the following description should be read carefully.

SIDE EFFECT: *load* now serves the function of both *fasl* and the old *load*. *Load* will search a user defined search path for a lisp source or object file with the filename s_filename (with the extension .l or .o added as appropriate). The search path which *load* uses is the value of *(status load-search-path)*. The default is (|.| /usr/lib/lisp) which means look in the current directory first and then /usr/lib/lisp. The file which *load* looks for depends on the last two characters of s_filename. If s_filename ends with ".l" then *load* will only look for a file name s_filename and will assume that this is a FRANZ LISP source file. If s_filename ends with ".o" then *load* will only look for a file named s_filename and will assume that this is a FRANZ LISP object file to be *fasl*ed in. Otherwise, *load* will first look for s_filename.o, then s_filename.l and finally s_filename itself. If it finds s_filename.o it will assume that this is an object file, otherwise it will assume that it is a source file. An object file is loaded using *fasl* and a source file is loaded by reading and evaluating each form in the file. The optional arguments st_map and g_warn are passed to *fasl* should *fasl* be called.

NOTE: *load* requires a port to open the file s_filename. It then lambda binds the symbol piport to this port and reads and evaluates the forms.

(makereadtable ['s_flag])

WHERE: if s_flag is not present it is assumed to be nil.

RETURNS:a readtable equal to the original readtable if s_flag is non-null, or else equal to the current readtable. See chapter 7 for a description of readtables and their uses.

(msg [l_option ...] ['g_msg ...])

NOTE: This function is intended for printing short messages. Any of the arguments or options presented can be used any number of times, in any order. The messages themselves (g_msg) are evaluated, and then they are transmitted to *patom*. Typically, they are strings, which evaluate to themselves. The options are interpreted specially:

msg Option Summary

(P p_portname)	causes subsequent output to go to the port p_portname (port should be opened previously)
B	print a single blank.
(B 'n_b)	evaluate n_b and print that many blanks.
N	print a single by calling *terpr*.
(N 'n_n)	evaluate n_n and transmit that many newlines to the stream.
D	*drain* the current port.

(nwritn ['p_port])

RETURNS: the number of characters in the buffer of the given port but not yet written out to the file or device. The buffer is flushed automatically when filled, or when *terpr* is called.

(outfile 's_filename ['st_type])

RETURNS: a port or nil

SIDE EFFECT: this opens a port to write s_filename. If st_type is given and if it is a symbol or string whose name begins with 'a', then the file will be opened in append mode, that is the current contents will not be lost and the next data will be written at the end of the file. Otherwise, the file opened is truncated by *outfile* if it existed beforehand. If there are no free ports, outfile returns nil. If one cannot write on s_filename, an error is signalled.

(patom 'g_exp ['p_port])

RETURNS: g_exp

SIDE EFFECT: g_exp is printed to the given port or the default port. If g_exp is a symbol or string, the print name is printed without any escape characters around special characters in the print name. If g_exp is a list then *patom* has the same effect as *print*.

(pntlen 'xfs_arg)

RETURNS:the number of characters needed to print xfs_arg.

(portp 'g_arg)

RETURNS:t iff g_arg is a port.

(pp [l_option] s_name1 ...**)**

RETURNS:t

SIDE EFFECT: If s_name*i* has a function binding, it is pretty-printed, otherwise if s_name*i* has a value then that is pretty-printed. Normally the output of the pretty-printer goes to the standard output port poport. The options allow you to redirect it.

PP Option Summary

(F s_filename)	direct future printing to s_filename
(P p_portname)	causes output to go to the port p_portname (port should be opened previously)
(E g_expression)	evaluate g_expression and don't print

(princ 'g_arg ['p_port])

EQUIVALENT TO:patom.

(print 'g_arg ['p_port])

RETURNS:nil

SIDE EFFECT: prints g_arg on the port p_port or the default port.

(probef 'st_file)

RETURNS:t iff the file st_file exists.

NOTE: Just because it exists doesn't mean you can read it.

(pp-form 'g_form ['p_port])

RETURNS:t

SIDE EFFECT: g_form is pretty-printed to the port p_port (or poport if p_port is not given). This is the function which *pp* uses. *pp-form* does not look for function definitions or values of variables, it just prints out the form it is given.

NOTE: This is useful as a top-level-printer, c.f. *top-level* in Chapter 6.

(ratom ['p_port ['g_eof]])

RETURNS: the next atom read from the given or default port. On end of file, g_eof (default nil) is returned.

(read ['p_port ['g_eof]])

RETURNS: the next lisp expression read from the given or default port. On end of file, g_eof (default nil) is returned.

NOTE: An error will occur if the reader is given an ill formed expression. The most common error is too many right parentheses (note that this is not considered an error in Maclisp).

(readc ['p_port ['g_eof]])

RETURNS: the next character read from the given or default port. On end of file, g_eof (default nil) is returned.

(readlist 'l_arg)

RETURNS: the lisp expression read from the list of characters in l_arg.

(removeaddress 's_name1 ['s_name2 ...])

RETURNS: nil

SIDE EFFECT: the entries for the s_name*i* in the Lisp symbol table are removed. This is useful if you wish to *cfasl* or *ffasl* in a file twice, since it is illegal for a symbol in the file you are loading to already exist in the lisp symbol table.

(resetio)

RETURNS: nil

SIDE EFFECT: all ports except the standard input, output and error are closed.

(setsyntax 's_symbol 's_synclass ['ls_func])

RETURNS: t

SIDE EFFECT: this sets the code for s_symbol to sx_code in the current readtable. If s_synclass is *macro* or *splicing* then ls_func is the associated function. See Chapter 7 on the reader for more details.

(sload 's_file)

SIDE EFFECT: the file s_file (in the current directory) is opened for reading and each form is read, printed and evaluated. If the form is recognizable as a function definition, only its name will be printed, otherwise the whole form is printed.

NOTE: This function is useful when a file refuses to load because of a syntax error and you would like to narrow down where the error is.

(tab 'x_col ['p_port])

SIDE EFFECT: enough spaces are printed to put the cursor on column x_col. If the cursor is beyond x_col to start with, a *terpr* is done first.

(terpr ['p_port])

RETURNS:nil

SIDE EFFECT: a terminate line character sequence is sent to the given port or the default port. This will also drain the port.

(terpri ['p_port])

EQUIVALENT TO:terpr.

(tilde-expand 'st_filename)

RETURNS:a symbol whose pname is the tilde-expansion of the argument, (as discussed at the beginning of this chapter). If the argument does not begin with a tilde, the argument itself is returned.

(tyi ['p_port])

RETURNS:the fixnum representation of the next character read. On end of file, -1 is returned.

(tyipeek ['p_port])

RETURNS:the fixnum representation of the next character to be read.

NOTE: This does not actually read the character, it just peeks at it.

(tyo 'x_char ['p_port])

RETURNS:x_char.

SIDE EFFECT: the character whose fixnum representation is x_code, is printed as a on the given output port or the default output port.

(untyi 'x_char ['p_port])

SIDE EFFECT: x_char is put back in the input buffer so a subsequent *tyi* or *read* will read it first.

NOTE: a maximum of one character may be put back.

(username-to-dir 'st_name)

RETURNS:the home directory of the given user. The result is stored, to avoid unnecessarily searching the password file.

(zapline)

RETURNS: nil

SIDE EFFECT: all characters up to and including the line termination character are read and discarded from the last port used for input.

NOTE: this is used as the macro function for the semicolon character when it acts as a comment character.

CHAPTER 6

System Functions

This chapter describes the functions used to interact with internal components of the Lisp system and operating system.

(allocate 's_type 'x_pages)

WHERE: s_type is one of the FRANZ LISP data types described in §1.3.

RETURNS:x_pages.

SIDE EFFECT: FRANZ LISP attempts to allocate x_pages of type s_type. If there aren't x_pages of memory left, no space will be allocated and an error will occur. The storage that is allocated is not given to the caller, instead it is added to the free storage list of s_type. The functions *segment* and *small-segment* allocate blocks of storage and return it to the caller.

(argv 'x_argnumb)

RETURNS:a symbol whose pname is the x_argnumb*th* argument (starting at 0) on the command line which invoked the current lisp.

NOTE: if x_argnumb is less than zero, a fixnum whose value is the number of arguments on the command line is returned. *(argv 0)* returns the name of the lisp you are running.

(baktrace)

RETURNS:nil

SIDE EFFECT: the lisp runtime stack is examined and the name of (most) of the functions currently in execution are printed, most active first.

NOTE: this will occasionally miss the names of compiled lisp functions due to incomplete information on the stack. If you are tracing compiled code, then *baktrace* won't be able to interpret the stack unless *(sstatus translink nil)* was done. See the function *showstack* for another way of printing the lisp runtime stack. This misspelling is from Maclisp.

(chdir 's_path)

RETURNS:t iff the system call succeeds.

SIDE EFFECT: the current directory set to s_path. Among other things, this will affect the default location where the input/output functions look for and create files.

NOTE: *chdir* follows the standard UNIX conventions, if s_path does not begin with a slash, the default path is changed to the current path with s_path appended. *Chdir* employs tilde-expansion (discussed in Chapter 5).

(command-line-args)

RETURNS: a list of the arguments typed on the command line, either to the lisp interpreter, or saved lisp dump, or application compiled with the autorun option (liszt -r).

(deref 'x_addr**)**

RETURNS: The contents of x_addr, when thought of as a longword memory location.

NOTE: This may be useful in constructing arguments to C functions out of 'dangerous' areas of memory.

(dumplisp s_name**)**

RETURNS: nil

SIDE EFFECT: the current lisp is dumped to the named file. When s_name is executed, you will be in a lisp in the same state as when the dumplisp was done.

NOTE: dumplisp will fail if one tries to write over the current running file. UNIX does not allow you to modify the file you are running.

(eval-when l_time g_exp1 ...**)**

SIDE EFFECT: l_time may contain any combination of the symbols *load*, *eval*, and *compile*. The effects of load and compile is discussed in §12.3.2.1 compiler. If eval is present however, this simply means that the expressions g_exp1 and so on are evaluated from left to right. If eval is not present, the forms are not evaluated.

(exit ['x_code]**)**

RETURNS: nothing (it never returns).

SIDE EFFECT: the lisp system dies with exit code x_code or 0 if x_code is not specified.

(fake 'x_addr**)**

RETURNS: the lisp object at address x_addr.

NOTE: This is intended to be used by people debugging the lisp system.

(fork)

RETURNS: nil to the child process and the process number of the child to the parent.

SIDE EFFECT: A copy of the current lisp system is made in memory and both lisp systems now begin to run. This function can be used interactively to temporarily save the state of Lisp (as shown below), but you must be careful that only one of the lisp's interacts with the terminal after the fork. The *wait* function is useful for this.

```
-> (setq foo 'bar)          ;; set a variable
bar
-> (cond ((fork)(wait)))    ;; duplicate the lisp system and
nil                                 ;; make the parent wait
-> foo                             ;; check the value of the variable
bar
-> (setq foo 'baz)          ;; give it a new value
baz
-> foo                             ;; make sure it worked
baz
-> (exit)                   ;; exit the child
(5274 . 0)                          ;; the wait function returns this
-> foo                             ;; we check to make sure parent was
bar                                 ;; not modified.
```

(gc)

RETURNS: nil

SIDE EFFECT: this causes a garbage collection.

NOTE: The function *gcafter* is not called automatically after this function finishes. Normally the user doesn't have to call *gc* since garbage collection occurs automatically whenever internal free lists are exhausted.

(gcafter s_type**)**

WHERE: s_type is one of the FRANZ LISP data types listed in §1.3.

NOTE: this function is called by the garbage collector after a garbage collection which was caused by running out of data type s_type. This function should determine if more space need be allocated and if so should allocate it. There is a default gcafter function but users who want control over space allocation can define their own -- but note that it must be an nlambda.

(getenv 's_name**)**

RETURNS: a symbol whose pname is the value of s_name in the current UNIX environment. If s_name doesn't exist in the current environment, a symbol with a null pname is returned.

(hashtabstat)

RETURNS: a list of fixnums representing the number of symbols in each bucket of the oblist.

NOTE: the oblist is stored a hash table of buckets. Ideally there would be the same number of symbols in each bucket.

(help [sx_arg])

SIDE EFFECT: If sx_arg is a symbol then the portion of this manual beginning with the description of sx_arg is printed on the terminal. If sx_arg is a fixnum or the name of one of the appendicies, that chapter or appendix is printed on the terminal. If no argument is provided, *help* prints the options that it recognizes. The program 'more' is used to print the manual on the terminal; it will stop after each page and will continue after the space key is pressed.

(include s_filename)

RETURNS:nil

SIDE EFFECT: The given filename is *load*ed into the lisp.

NOTE: this is similar to load except the argument is not evaluated. Include means something special to the compiler.

(include-if 'g_predicate s_filename)

RETURNS:nil

SIDE EFFECT: This has the same effect as include, but is only actuated if the predicate is non-nil.

(includef 's_filename)

RETURNS:nil

SIDE EFFECT: this is the same as *include* except the argument is evaluated.

(includef-if 'g_predicate s_filename)

RETURNS:nil

SIDE EFFECT: This has the same effect as includef, but is only actuated if the predicate is non-nil.

(maknum 'g_arg)

RETURNS:the address of its argument converted into a fixnum.

(monitor ['xs_maxaddr])

RETURNS:t

SIDE EFFECT: If xs_maxaddr is t then profiling of the entire lisp system is begun. If xs_maxaddr is a fixnum then profiling is done only up to address xs_maxaddr. If xs_maxaddr is not given, then profiling is stopped and the data obtained is written to the file 'mon.out' where it can be analyzed with the UNIX 'prof' program.

NOTE: this function only works if the lisp system has been compiled in a special way, otherwise, an error is invoked.

(opval 's_arg ['g_newval])

RETURNS: the value associated with s_arg before the call.

SIDE EFFECT: If g_newval is specified, the value associated with s_arg is changed to g_newval.

NOTE: *opval* keeps track of storage allocation. If s_arg is one of the data types then *opval* will return a list of three fixnums representing the number of items of that type in use, the number of pages allocated and the number of items of that type per page. You should never try to change the value *opval* associates with a data type using *opval.*
If s_arg is *pagelimit* then *opval* will return (and set if g_newval is given) the maximum amount of lisp data pages it will allocate. This limit should remain small unless you know your program requires lots of space as this limit will catch programs in infinite loops which gobble up memory.

(*process 'st_command ['g_readp ['g_writep]])

RETURNS: either a fixnum if one argument is given, or a list of two ports and a fixnum if two or three arguments are given.

NOTE: **process* starts another process by passing st_command to the shell (it first tries /bin/csh, then it tries /bin/sh if /bin/csh doesn't exist). If only one argument is given to **process*, **process* waits for the new process to die and then returns the exit code of the new process. If more two or three arguments are given, **process* starts the process and then returns a list which, depending on the value of g_readp and g_writep, may contain i/o ports for communcating with the new process. If g_writep is non-null, then a port will be created which the lisp program can use to send characters to the new process. If g_readp is non-null, then a port will be created which the lisp program can use to read characters from the new process. The value returned by **process* is (readport writeport pid) where readport and writeport are either nil or a port based on the value of g_readp and g_writep. Pid is the process id of the new process. Since it is hard to remember the order of g_readp and g_writep, the functions **process-send* and **process-receive* were written to perform the common functions.

(*process-receive 'st_command)

RETURNS: a port which can be read.

SIDE EFFECT: The command st_command is given to the shell and it is started running in the background. The output of that command is available for reading via the port returned. The input of the command process is set to /dev/null.

(*process-send 'st_command)

RETURNS: a port which can be written to.

SIDE EFFECT: The command st_command is given to the shell and it is started runing in the background. The lisp program can provide input for that command by sending characters to the port returned by this function. The output of the command process is set to /dev/null.

(process s_pgrm [s_frompipe s_topipe])

RETURNS: if the optional arguments are not present a fixnum which is the exit code when s_prgm dies. If the optional arguments are present, it returns a fixnum which is the process id of the child.

NOTE: This command is obsolete. New programs should use one of the **process* commands given above.

SIDE EFFECT: If s_frompipe and s_topipe are given, they are bound to ports which are pipes which direct characters from FRANZ LISP to the new process and to FRANZ LISP from the new process respectively. *Process* forks a process named s_prgm and waits for it to die iff there are no pipe arguments given.

(ptime)

RETURNS: a list of two elements. The first is the amount of processor time used by the lisp system so far, and the second is the amount of time used by the garbage collector so far.

NOTE: the time is measured in those units used by the *times*(2) system call, usually 60*th*s of a second. The first number includes the second number. The amount of time used by garbage collection is not recorded until the first call to ptime. This is done to prevent overhead when the user is not interested in garbage collection times.

(reset)

SIDE EFFECT: the lisp runtime stack is cleared and the system restarts at the top level by executing a *(funcall top-level nil)*.

(restorelisp 's_name)

SIDE EFFECT: this reads in file s_name (which was created by *savelisp*) and then does a *(reset)*.

NOTE: This is only used on VMS systems where *dumplisp* cannot be used.

(retbrk ['x_level])

WHERE: x_level is a small integer of either sign.

SIDE EFFECT: The default error handler keeps a notion of the current level of the error caught. If x_level is negative, control is thrown to this default error handler whose level is that many less than the present, or to *top-level* if there aren't enough. If x_level is non-negative, control is passed to the handler at that level. If x_level is not present, the value -1 is taken by default.

(*rset 'g_flag)

RETURNS:g_flag

SIDE EFFECT: If g_flag is non nil then the lisp system will maintain extra information about calls to *eval* and *funcall*. This record keeping slows down the evaluation but this is required for the functions *evalhook*, *funcallhook*, and *evalframe* to work. To debug compiled lisp code the transfer tables should be unlinked: *(sstatus translink nil)*

(savelisp 's_name)

RETURNS:t

SIDE EFFECT: the state of the Lisp system is saved in the file s_name. It can be read in by *restorelisp*.

NOTE: This is only used on VMS systems where *dumplisp* cannot be used.

(segment 's_type 'x_size)

WHERE: s_type is one of the data types given in §1.3

RETURNS:a segment of contiguous lispvals of type s_type.

NOTE: In reality, *segment* returns a new data cell of type s_type and allocates space for x_size - 1 more s_type's beyond the one returned. *Segment* always allocates new space and does so in 512 byte chunks. If you ask for 2 fixnums, segment will actually allocate 128 of them thus wasting 126 fixnums. The function *small-segment* is a smarter space allocator and should be used whenever possible.

(shell)

RETURNS:the exit code of the shell when it dies.

SIDE EFFECT: this forks a new shell and returns when the shell dies.

(showstack)

RETURNS:nil

SIDE EFFECT: all forms currently in evaluation are printed, beginning with the most recent. For compiled code the most that showstack will show is the function name and it may miss some functions.

(signal 'x_signum 's_name)

RETURNS:nil if no previous call to signal has been made, or the previously installed s_name.

SIDE EFFECT: this declares that the function named s_name will handle the signal number x_signum. If s_name is nil, the signal is ignored. Presently only four UNIX signals are caught. They and their numbers are: Interrupt(2), Floating exception(8), Alarm(14), and Hang-up(1).

(sizeof 'g_arg)

RETURNS: the number of bytes required to store one object of type g_arg, encoded as a fixnum.

(small-segment 's_type 'x_cells)

WHERE: s_type is one of fixnum, flonum and value.

RETURNS: a segment of x_cells data objects of type s_type.

SIDE EFFECT: This may call *segment* to allocate new space or it may be able to fill the request on a page already allocated. The value returned by *small-segment* is usually stored in the data subpart of an array object.

(sstatus g_type g_val)

RETURNS: g_val

SIDE EFFECT: If g_type is not one of the special sstatus codes described in the next few pages this simply sets g_val as the value of status type g_type in the system status property list.

(sstatus appendmap g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when *fasl* is told to create a load map, it will append to the file name given in the *fasl* command, rather than creating a new map file. The initial value is nil.

(sstatus automatic-reset g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when an error occurs which no one wants to handle, a *reset* will be done instead of entering a primitive internal break loop. The initial value is t.

(sstatus chainatom g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non nil and a *car* or *cdr* of a symbol is done, then nil will be returned instead of an error being signaled. This only affects the interpreter, not the compiler. The initial value is nil.

(sstatus dumpcore g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is nil, FRANZ LISP tells UNIX that a segmentation violation or bus error should cause a core dump. If g_val is non nil then FRANZ LISP will catch those errors and print a message advising the user to reset.

NOTE: The initial value for this flag is nil, and only those knowledgeable of the innards of the lisp system should ever set this flag non nil.

(sstatus dumpmode x_val**)**

RETURNS:x_val

SIDE EFFECT: All subsequent *dumplisp*'s will be done in mode x_val. x_val may be either 413 or 410 (decimal).

NOTE: the advantage of mode 413 is that the dumped Lisp can be demand paged in when first started, which will make it start faster and disrupt other users less. The initial value is 413.

(sstatus evalhook g_val**)**

RETURNS:g_val

SIDE EFFECT: When g_val is non nil, this enables the evalhook and funcallhook traps in the evaluator. See §14.4 for more details.

(sstatus feature g_val**)**

RETURNS:g_val

SIDE EFFECT: g_val is added to the *(status features)* list,

(sstatus gcstrings g_val**)**

RETURNS:g_val

SIDE EFFECT: if g_val is non-null, and if string garbage collection was enabled when the lisp system was compiled, string space will be garbage collected.

NOTE: the default value for this is nil since in most applications garbage collecting strings is a waste of time.

(sstatus ignoreeof g_val**)**

RETURNS:g_val

SIDE EFFECT: If g_val is non-null when an end of file (CNTL-D on UNIX) is typed to the standard top-level interpreter, it will be ignored rather then cause the lisp system to exit. If the the standard input is a file or pipe then this has no effect, an EOF will always cause lisp to exit. The initial value is nil.

(sstatus nofeature g_val**)**

RETURNS:g_val

SIDE EFFECT: g_val is removed from the status features list if it was present.

(sstatus translink g_val**)**

RETURNS:g_val

SIDE EFFECT: If g_val is nil then all transfer tables are cleared and further calls through the transfer table will not cause the fast links to be set up. If g_val is the symbol *on* then all possible transfer table entries will be linked and the flag will be set to cause fast links to be set up dynamically. Otherwise all that is done is to set the flag to cause fast links to be set up dynamically. The initial value is nil.

NOTE: For a discussion of transfer tables, see §12.8.

(sstatus uctolc g_val)

RETURNS:g_val

SIDE EFFECT: If g_val is not nil then all unescaped capital letters in symbols read by the reader will be converted to lower case.

NOTE: This allows FRANZ LISP to be compatible with single case lisp systems (e.g. Maclisp, Interlisp and UCILisp).

(status g_code)

RETURNS:the value associated with the status code g_code if g_code is not one of the special cases given below

(status ctime)

RETURNS:a symbol whose print name is the current time and date.

EXAMPLE:*(status ctime)* = |Sun Jun 29 16:51:26 1980|

NOTE: This has been made obsolete by *time-string*, described below.

(status feature g_val)

RETURNS:t iff g_val is in the status features list.

(status features)

RETURNS:the value of the features code, which is a list of features which are present in this system. You add to this list with *(sstatus feature 'g_val)* and test if feature g_feat is present with *(status feature 'g_feat)*.

(status isatty)

RETURNS:t iff the standard input is a terminal.

(status localtime)

RETURNS:a list of fixnums representing the current time.

EXAMPLE:*(status localtime)* = (3 51 13 31 6 81 5 211 1)
means 3*rd* second, 51*st* minute, 13*th* hour (1 p.m), 31*st* day, month 6 (0 = January), year 81 (0 = 1900), day of the week 5 (0 = Sunday), 211*th* day of the year and daylight savings time is in effect.

(status syntax s_char)

NOTE: This function should not be used. See the description of *getsyntax* (in Chapter 7) for a replacement.

(status undeffunc)

RETURNS: a list of all functions which transfer table entries point to but which are not defined at this point.

NOTE: Some of the undefined functions listed could be arrays which have yet to be created.

(status version)

RETURNS: a string which is the current lisp version name.

EXAMPLE: *(status version)* = "Franz Lisp, Opus 38.61"

(syscall 'x_index ['xst_arg1 ...])

RETURNS: the result of issuing the UNIX system call number x_index with arguments xst_arg*i*.

NOTE: The UNIX system calls are described in section 2 of the UNIX Programmer's manual. If xst_arg*i* is a fixnum, then its value is passed as an argument, if it is a symbol then its pname is passed and finally if it is a string then the string itself is passed as an argument. Some useful syscalls are:
(syscall 20) returns process id.
(syscall 13) returns the number of seconds since Jan 1, 1970.
(syscall 10 'foo) will unlink (delete) the file foo.

(sys:access 'st_filename 'x_mode)
(sys:chmod 'st_filename 'x_mode)
(sys:gethostname)
(sys:getpid)
(sys:getpwnam 'st_username)
(sys:link 'st_oldfilename 'st_newfilename)
(sys:time)
(sys:unlink 'st_filename)

NOTE: We have been warned that the actual system call numbers may vary among different UNIX systems. Users concerned about portability may wish to use this group of functions. Another advantage is that tilde-expansion is performed on all filename arguments. These functions do what is described in the system call section of your UNIX manual.

sys:getpwnam returns a vector of four entries from the password file, being the user name, user id, group id, and home directory.

(time-string ['x_seconds])

RETURNS: an ascii string giving the time and date which was x_seconds after UNIX's idea of creation (Midnight, Jan 1, 1970 GMT). If no argument is given, time-string returns the current date. This supplants *(status ctime)*, and may be used to make the results of *filestat* more intelligible.

(top-level)

RETURNS: nothing (it never returns)

NOTE: This function is the top-level read-eval-print loop. It never returns any value. Its main utility is that if you redefine it, and do a (reset) then the redefined (top-level) is then invoked. The default top-level for Franz, allow one to specify his own printer or reader, by binding the symbols **top-level-printer** and **top-level-reader**. One can let the default top-level do most of the drudgery in catching *reset*'s, and reading in .lisprc files, by binding the symbol **user-top-level**, to a routine that concerns itself only with the read-eval-print loop.

(wait)

RETURNS: a dotted pair *(processid . status)* when the next child process dies.

CHAPTER 7

The Lisp Reader

7.1. Introduction

The *read* function is responsible for converting a stream of characters into a Lisp expression. *Read* is table driven and the table it uses is called a *readtable.* The *print* function does the inverse of *read*; it converts a Lisp expression into a stream of characters. Typically the conversion is done in such a way that if that stream of characters were read by *read*, the result would be an expression equal to the one *print* was given. *Print* must also refer to the readtable in order to determine how to format its output. The *explode* function, which returns a list of characters rather than printing them, must also refer to the readtable.

A readtable is created with the *makereadtable* function, modified with the *setsyntax* function and interrogated with the *getsyntax* function. The structure of a readtable is hidden from the user - a readtable should only be manipulated with the three functions mentioned above.

There is one distinguished readtable called the *current readtable* whose value determines what *read*, *print* and *explode* do. The current readtable is the value of the symbol *readtable.* Thus it is possible to rapidly change the current syntax by lambda binding a different readtable to the symbol *readtable.* When the binding is undone, the syntax reverts to its old form.

7.2. Syntax Classes

The readtable describes how each of the 128 ascii characters should be treated by the reader and printer. Each character belongs to a *syntax class* which has three properties:

character class -
: Tells what the reader should do when it sees this character. There are a large number of character classes. They are described below.

separator -
: Most types of tokens the reader constructs are one character long. Four token types have an arbitrary length: number (1234), symbol print name (franz), escaped symbol print name (|franz|), and string ("franz"). The reader can easily determine when it has come to the end of one of the last two types: it just looks for the matching delimiter (| or "). When the reader is reading a number or symbol print name, it stops reading when it comes to a character with the *separator* property. The separator character is pushed back into the input stream and will be the first character read when the reader is called again.

escape -
: Tells the printer when to put escapes in front of, or around, a symbol whose print

name contains this character. There are three possibilities: always escape a symbol with this character in it, only escape a symbol if this is the only character in the symbol, and only escape a symbol if this is the first character in the symbol. [note: The printer will always escape a symbol which, if printed out, would look like a valid number.]

When the Lisp system is built, Lisp code is added to a C-coded kernel and the result becomes the standard lisp system. The readtable present in the C-coded kernel, called the *raw readtable*, contains the bare necessities for reading in Lisp code. During the construction of the complete Lisp system, a copy is made of the raw readtable and then the copy is modified by adding macro characters. The result is what is called the *standard readtable*. When a new readtable is created with *makereadtable*, a copy is made of either the raw readtable or the current readtable (which is likely to be the standard readtable).

7.3. Reader Operations

The reader has a very simple algorithm. It is either *scanning* for a token, *collecting* a token, or *processing* a token. Scanning involves reading characters and throwing away those which don't start tokens (such as blanks and tabs). Collecting means gathering the characters which make up a token into a buffer. Processing may involve creating symbols, strings, lists, fixnums, bignums or flonums or calling a user written function called a character macro.

The components of the syntax class determine when the reader switches between the scanning, collecting and processing states. The reader will continue scanning as long as the character class of the characters it reads is *cseparator*. When it reads a character whose character class is not *cseparator* it stores that character in its buffer and begins the collecting phase.

If the character class of that first character is *ccharacter*, *cnumber*, *cperiod*, or *csign*. then it will continue collecting until it runs into a character whose syntax class has the *separator* property. (That last character will be pushed back into the input buffer and will be the first character read next time.) Now the reader goes into the processing phase, checking to see if the token it read is a number or symbol. It is important to note that after the first character is collected the component of the syntax class which tells the reader to stop collecting is the *separator* property, not the character class.

If the character class of the character which stopped the scanning is not *ccharacter*, *cnumber*, *cperiod*, or *csign*. then the reader processes that character immediately. The character classes *csingle-macro*, *csingle-splicing-macro*, and *csingle-infix-macro* will act like *ccharacter* if the following token is not a *separator*. The processing which is done for a given character class is described in detail in the next section.

7.4. Character Classes

ccharacter raw readtable:A-Z a-z ^H !#$%&*,/:;<=>?@^_`{}~
standard readtable:A-Z a-z ^H !$%&*/:;<=>?@^_{}~

A normal character.

cnumber raw readtable:0-9
standard readtable:0-9

This type is a digit. The syntax for an integer (fixnum or bignum) is a string of *cnumber* characters optionally followed by a *cperiod*. If the digits are not followed by a *cperiod*, then they are interpreted in base *ibase* which must be eight or ten. The syntax for a floating point number is either zero or more *cnumber*'s followed by a *cperiod* and then followed by one or more *cnumber*'s. A floating point number may also be an integer or floating point number followed by 'e' or 'd', an optional '+' or '–' and then zero or more *cnumber*'s.

csign raw readtable:+–
standard readtable:+–

A leading sign for a number. No other characters should be given this class.

cleft-paren raw readtable:(
standard readtable:(

A left parenthesis. Tells the reader to begin forming a list.

cright-paren raw readtable:)
standard readtable:)

A right parenthesis. Tells the reader that it has reached the end of a list.

cleft-bracket raw readtable:[
standard readtable:[

A left bracket. Tells the reader that it should begin forming a list. See the description of *cright-bracket* for the difference between cleft-bracket and cleft-paren.

cright-bracket raw readtable:]
standard readtable:]

A right bracket. A *cright-bracket* finishes the formation of the current list and all enclosing lists until it finds one which begins with a *cleft-bracket* or until it reaches the top level list.

cperiod raw readtable:.
standard readtable:.

The period is used to separate element of a cons cell [e.g. (a . (b . nil)) is the same as (a b)]. *cperiod* is also used in numbers as described above.

cseparator raw readtable:^I-^M esc space
standard readtable:^I-^M esc space

Separates tokens. When the reader is scanning, these character are passed over. Note: there is a difference between the *cseparator* character class and the *separator* property of a syntax class.

csingle-quote raw readtable:'
standard readtable:'

This causes *read* to be called recursively and the list (quote <value read>) to be returned.

csymbol-delimiter raw readtable:|
standard readtable:|

This causes the reader to begin collecting characters and to stop only when another identical *csymbol-delimiter* is seen. The only way to escape a *csymbol-delimiter* within a symbol name is with a *cescape* character. The collected characters are converted into a string which becomes the print name of a symbol. If a symbol with an identical print name already exists, then the allocation is not done, rather the existing symbol is used.

cescape raw readtable:\
standard readtable:\

This causes the next character to read in to be treated as a **vcharacter**. A character whose syntax class is **vcharacter** has a character class *ccharacter* and does not have the *separator* property so it will not separate symbols.

cstring-delimiter raw readtable:"
standard readtable:"

This is the same as *csymbol-delimiter* except the result is returned as a string instead of a symbol.

csingle-character-symbol raw readtable:none
standard readtable:none

This returns a symbol whose print name is the the single character which has been collected.

cmacro raw readtable:none
standard readtable:',

The reader calls the macro function associated with this character and the current readtable, passing it no arguments. The result of the macro is added to the structure the reader is building, just as if that form were directly read by the reader. More details on macros are provided below.

csplicing-macro raw readtable:none
standard readtable:#;

A *csplicing-macro* differs from a *cmacro* in the way the result is incorporated in the structure the reader is building. A *csplicing-macro* must return a list of forms (possibly empty). The reader acts as if it read each element of the list itself without the surrounding parenthesis.

csingle-macro raw readtable:none
standard readtable:none

This causes to reader to check the next character. If it is a *cseparator* then this acts like a *cmacro*. Otherwise, it acts like a *ccharacter*.

csingle-splicing-macro raw readtable:none
standard readtable:none

This is triggered like a *csingle-macro* however the result is spliced in like a *csplicing-macro.*

cinfix-macro raw readtable:none
standard readtable:none

This is differs from a *cmacro* in that the macro function is passed a form representing what the reader has read so far. The result of the macro replaces what the reader had read so far.

csingle-infix-macro raw readtable:none
standard readtable:none

This differs from the *cinfix-macro* in that the macro will only be triggered if the character following the *csingle-infix-macro* character is a *cseparator.*

cillegal raw readtable:^@-^G^N-^Z^\-^_rubout
standard readtable:^@-^G^N-^Z^\-^_rubout

The characters cause the reader to signal an error if read.

7.5. Syntax Classes

The readtable maps each character into a syntax class. The syntax class contains three pieces of information: the character class, whether this is a separator, and the escape properties. The first two properties are used by the reader, the last by the printer (and *explode*). The initial lisp system has the following syntax classes defined. The user may add syntax classes with *add-syntax-class.* For each syntax class, we list the properties of the class and which characters have this syntax class by default. More information about each syntax class can be found under the description of the syntax class's character class.

vcharacter raw readtable:A-Z a-z ^H !#$%&*,/:;<=>?@^_`{}~
ccharacter standard readtable:A-Z a-z ^H !$%&*/:;<=>?@^_{}~

vnumber raw readtable:0-9
cnumber standard readtable:0-9

vsign raw readtable:+-
csign standard readtable:+-

vleft-paren
cleft-paren
escape-always
raw readtable:(
standard readtable:(

vright-paren
cright-paren
escape-always
raw readtable:)
standard readtable:)

vleft-bracket
cleft-bracket
escape-always
raw readtable:[
standard readtable:[

vright-bracket
cright-bracket
escape-always
raw readtable:]
standard readtable:]

vperiod
cperiod
escape-when-unique
raw readtable:.
standard readtable:.

vseparator
cseparator
escape-always
raw readtable:^I-^M esc space
standard readtable:^I-^M esc space

vsingle-quote
csingle-quote
escape-always
raw readtable:'
standard readtable:'

vsymbol-delimiter
csingle-delimiter
escape-always
raw readtable:|
standard readtable:|

vescape
cescape
escape-always
raw readtable:\
standard readtable:\

vstring-delimiter
cstring-delimiter
escape-always
raw readtable:"
standard readtable:"

vsingle-character-symbol
csingle-character-symbol
separator
raw readtable:none
standard readtable:none

vmacro raw readtable:none
cmacro standard readtable:`,
escape-always

vsplicing-macro raw readtable:none
csplicing-macro standard readtable:#;
escape-always

vsingle-macro raw readtable:none
csingle-macro standard readtable:none
escape-when-unique

vsingle-splicing-macro raw readtable:none
csingle-splicing-macro standard readtable:none
escape-when-unique

vinfix-macro raw readtable:none
cinfix-macro standard readtable:none
escape-always

vsingle-infix-macro raw readtable:none
csingle-infix-macro standard readtable:none
escape-when-unique

villegal raw readtable:^@-^G^N-^Z^\-^_rubout
cillegal standard readtable:^@-^G^N-^Z^\-^_rubout
escape-always

7.6. Character Macros

Character macros are user written functions which are executed during the reading process. The value returned by a character macro may or may not be used by the reader, depending on the type of macro and the value returned. Character macros are always attached to a single character with the *setsyntax* function.

7.6.1. Types There are three types of character macros: normal, splicing and infix. These types differ in the arguments they are given or in what is done with the result they return.

7.6.1.1. Normal

A normal macro is passed no arguments. The value returned by a normal macro is simply used by the reader as if it had read the value itself. Here is an example of a macro which returns the abbreviation for a given state.

```
->(defun stateabbrev nil
      (cdr (assq (read) '((california . ca) (pennsylvania . pa)))))
stateabbrev
-> (setsyntax '\! 'vmacro 'stateabbrev)
t
-> '( ! california ! wyoming ! pennsylvania)
(ca nil pa)
```

Notice what happened to
! wyoming. Since it wasn't in the table, the associated function returned nil. The creator of the macro may have wanted to leave the list alone, in such a case, but couldn't with this type of reader macro. The splicing macro, described next, allows a character macro function to return a value that is ignored.

7.6.1.2. Splicing

The value returned from a splicing macro must be a list or nil. If the value is nil, then the value is ignored, otherwise the reader acts as if it read each object in the list. Usually the list only contains one element. If the reader is reading at the top level (i.e. not collecting elements of list), then it is illegal for a splicing macro to return more then one element in the list. The major advantage of a splicing macro over a normal macro is the ability of the splicing macro to return nothing. The comment character (usually ;) is a splicing macro bound to a function which reads to the end of the line and always returns nil. Here is the previous example written as a splicing macro

```
-> (defun stateabbrev nil
      ((lambda (value)
           (cond (value (list value))
                 (t nil)))
       (cdr (assq (read) '((california . ca) (pennsylvania . pa))))))
-> (setsyntax '! 'vsplicing-macro 'stateabbrev)
-> '(!pennsylvania ! foo !california)
(pa ca)
-> '!foo !bar !pennsylvania
pa
->
```

7.6.1.3. Infix

Infix macros are passed a *conc* structure representing what has been read so far. Briefly, a tconc structure is a single list cell whose car points to a list and whose cdr points to the last list cell in that list. The interpretation by the reader

of the value returned by an infix macro depends on whether the macro is called while the reader is constructing a list or whether it is called at the top level of the reader. If the macro is called while a list is being constructed, then the value returned should be a tconc structure. The car of that structure replaces the list of elements that the reader has been collecting. If the macro is called at top level, then it will be passed the value nil, and the value it returns should either be nil or a tconc structure. If the macro returns nil, then the value is ignored and the reader continues to read. If the macro returns a tconc structure of one element (i.e. whose car is a list of one element), then that single element is returned as the value of *read*. If the macro returns a tconc structure of more than one element, then that list of elements is returned as the value of read.

```
-> (defun plusop (x)
      (cond ((null x) (tconc nil '+))
            (t (lconc nil (list 'plus (caar x) (read))))))

plusop
-> (setsyntax '+ 'vinfix-macro 'plusop)
t
-> '(a + b)
(plus a b)
-> '+
|+|
->
```

7.6.2. Invocations

There are three different circumstances in which you would like a macro function to be triggered.

Always -
Whenever the macro character is seen, the macro should be invoked. This is accomplished by using the character classes *cmacro*, *csplicing-macro*, or *cinfix-macro*, and by using the *separator* property. The syntax classes **vmacro**, **vsplicing-macro**, and **vsingle-macro** are defined this way.

When first -
The macro should only be triggered when the macro character is the first character found after the scanning process. A syntax class for a *when first* macro would be defined using *cmacro*, *csplicing-macro*, or *cinfix-macro* and not including the *separator* property.

When unique -
The macro should only be triggered when the macro character is the only character collected in the token collection phase of the reader, i.e the macro character is preceeded by zero or more *cseparators* and followed by a *separator*. A syntax class for a *when unique* macro would be defined using *csingle-macro*, *csingle-splicing-macro*, or *csingle-infix-macro* and not including the *separator* property. The syntax classes so defined are **vsingle-macro**, **vsingle-splicing-macro**, and **vsingle-infix-macro**.

7.7. Functions

(setsyntax 's_symbol 's_synclass ['ls_func])

WHERE: ls_func is the name of a function or a lambda body.

RETURNS:t

SIDE EFFECT: S_symbol should be a symbol whose print name is only one character. The syntax class for that character is set to s_synclass in the current readtable. If s_synclass is a class that requires a character macro, then ls_func must be supplied.

NOTE: The symbolic syntax codes are new to Opus 38. For compatibility, s_synclass can be one of the fixnum syntax codes which appeared in older versions of the FRANZ LISP Manual. This compatibility is only temporary: existing code which uses the fixnum syntax codes should be converted.

(getsyntax 's_symbol)

RETURNS:the syntax class of the first character of s_symbol's print name. s_symbol's print name must be exactly one character long.

NOTE: This function is new to Opus 38. It supercedes *(status syntax)* which no longer exists.

(add-syntax-class 's_synclass 'l_properties)

RETURNS:s_synclass

SIDE EFFECT: Defines the syntax class s_synclass to have properties l_properties. The list l_properties should contain a character classes mentioned above. l_properties may contain one of the escape properties: *escape-always*, *escape-when-unique*, or *escape-when-first*. l_properties may contain the *separator* property. After a syntax class has been defined with *add-syntax-class*, the *setsyntax* function can be used to give characters that syntax class.

```
; Define a non-separating macro character.
; This type of macro character is used in UCI-Lisp, and
; it corresponds to a  FIRST MACRO in Interlisp
-> (add-syntax-class 'vuci-macro '(cmacro escape-when-first))
vuci-macro
->
```

CHAPTER 8

Functions, Fclosures, and Macros

8.1. valid function objects

There are many different objects which can occupy the function field of a symbol object. Table 8.1, on the following page, shows all of the possibilities, how to recognize them, and where to look for documentation.

8.2. functions

The basic Lisp function is the lambda function. When a lambda function is called, the actual arguments are evaluated from left to right and are lambda-bound to the formal parameters of the lambda function.

An nlambda function is usually used for functions which are invoked by the user at top level. Some built-in functions which evaluate their arguments in special ways are also nlambdas (e.g *cond*, *do*, *or*). When an nlambda function is called, the list of unevaluated arguments is lambda bound to the single formal parameter of the nlambda function.

Some programmers will use an nlambda function when they are not sure how many arguments will be passed. Then, the first thing the nlambda function does is map *eval* over the list of unevaluated arguments it has been passed. This is usually the wrong thing to do, as it will not work compiled if any of the arguments are local variables. The solution is to use a lexpr. When a lexpr function is called, the arguments are evaluated and a fixnum whose value is the number of arguments is lambda-bound to the single formal parameter of the lexpr function. The lexpr can then access the arguments using the *arg* function.

When a function is compiled, *special* declarations may be needed to preserve its behavior. An argument is not lambda-bound to the name of the corresponding formal parameter unless that formal parameter has been declared *special* (see §12.3.2.2).

Lambda and lexpr functions both compile into a binary object with a discipline of lambda. However, a compiled lexpr still acts like an interpreted lexpr.

8.3. macros

An important feature of Lisp is its ability to manipulate programs as data. As a result of this, most Lisp implementations have very powerful macro facilities. The Lisp language's macro facility can be used to incorporate popular features of the other languages into Lisp. For example, there are macro packages which allow one to create records (ala Pascal) and refer to elements of those records by the field names. The

informal name	object type	documentation
interpreted lambda function	list with *car* *eq* to lambda	8.2
interpreted nlambda function	list with *car* *eq* to nlambda	8.2
interpreted lexpr function	list with *car* *eq* to lexpr	8.2
interpreted macro	list with *car* *eq* to macro	8.3
fclosure	vector with *vprop* *eq* to fclosure	8.4
compiled lambda or lexpr function	binary with discipline *eq* to lambda	8.2
compiled nlambda function	binary with discipline *eq* to nlambda	8.2
compiled macro	binary with discipline *eq* to macro	8.3
foreign subroutine	binary with discipline of "subroutine"†	8.5
foreign function	binary with discipline of "function"†	8.5
foreign integer function	binary with discipline of "integer-function"†	8.5
foreign real function	binary with discipline of "real-function"†	8.5
foreign C function	binary with discipline of "c-function"†	8.5
foreign double function	binary with discipline of "double-c-function"†	8.5
foreign structure function	binary with discipline of "vector-c-function"†	8.5
array	array object	9

Table 8.1

struct package imported from Maclisp does this. Another popular use for macros is to create more readable control structures which expand into *cond*, *or* and *and*. One such example is the If macro. It allows you to write

```
(If (equal numb 0) then (print 'zero) (terpr)
 elseif (equal numb 1) then (print 'one) (terpr)
 else (print '|I give up|))
```

†**Only the first character of the string is significant (i.e "s" is ok for "subroutine")**

which expands to

```
(cond
   ((equal numb 0) (print 'zero) (terpr))
   ((equal numb 1) (print 'one) (terpr))
   (t (print '|I give up|)))
```

8.3.1. macro forms

A macro is a function which accepts a Lisp expression as input and returns another Lisp expression. The action the macro takes is called macro expansion. Here is a simple example:

```
-> (def first (macro (x) (cons 'car (cdr x))))
first
-> (first '(a b c))
a
-> (apply 'first '(first '(a b c)))
(car '(a b c))
```

The first input line defines a macro called *first*. Notice that the macro has one formal parameter, *x*. On the second input line, we ask the interpreter to evaluate *(first '(a b c))*. *Eval* sees that *first* has a function definition of type macro, so it evaluates *first*'s definition, passing to *first*, as an argument, the form *eval* itself was trying to evaluate: *(first '(a b c))*. The *first* macro chops off the car of the argument with *cdr*, cons' a *car* at the beginning of the list and returns *(car '(a b c))*, which *eval* evaluates. The value *a* is returned as the value of *(first '(a b c))*. Thus whenever *eval* tries to evaluate a list whose car has a macro definition it ends up doing (at least) two operations, the first of which is a call to the macro to let it macro expand the form, and the other is the evaluation of the result of the macro. The result of the macro may be yet another call to a macro, so *eval* may have to do even more evaluations until it can finally determine the value of an expression. One way to see how a macro will expand is to use *apply* as shown on the third input line above.

8.3.2. defmacro

The macro *defmacro* makes it easier to define macros because it allows you to name the arguments to the macro call. For example, suppose we find ourselves often writing code like *(setq stack (cons newelt stack)*. We could define a macro named *push* to do this for us. One way to define it is:

```
-> (def push
      (macro (x) (list 'setq (caddr x) (list 'cons (cadr x) (caddr x)))))
push
```

then *(push newelt stack)* will expand to the form mentioned above. The same macro written using defmacro would be:

```
-> (defmacro push (value stack)
```

```
  (list 'setq ,stack (list 'cons ,value ,stack)))
push
```

Defmacro allows you to name the arguments of the macro call, and makes the macro definition look more like a function definition.

8.3.3. the backquote character macro

The default syntax for FRANZ LISP has four characters with associated character macros. One is semicolon for comments. Two others are the backquote and comma which are used by the backquote character macro. The fourth is the sharp sign macro described in the next section.

The backquote macro is used to create lists where many of the elements are fixed (quoted). This makes it very useful for creating macro definitions. In the simplest case, a backquote acts just like a single quote:

```
->'(a b c d e)
(a b c d e)
```

If a comma precedes an element of a backquoted list then that element is evaluated and its value is put in the list.

```
->(setq d '(x y z))
(x y z)
->'(a b c ,d e)
(a b c (x y z) e)
```

If a comma followed by an at sign precedes an element in a backquoted list, then that element is evaluated and spliced into the list with *append*.

```
->'(a b c ,@d e)
(a b c x y z e)
```

Once a list begins with a backquote, the commas may appear anywhere in the list as this example shows:

```
->'(a b (c d ,(cdr d)) (e f (g h ,@(cddr d) ,@d)))
(a b (c d (y z)) (e f (g h z x y z)))
```

It is also possible and sometimes even useful to use the backquote macro within itself. As a final demonstration of the backquote macro, we shall define the first and push macros using all the power at our disposal: defmacro and the backquote macro.

```
->(defmacro first (list) '(car ,list))
first
->(defmacro push (value stack) '(setq ,stack (cons ,value ,stack)))
stack
```

8.3.4. sharp sign character macro

The sharp sign macro can perform a number of different functions at read time. The character directly following the sharp sign determines which function will be done, and following Lisp s-expressions may serve as arguments.

8.3.4.1. conditional inclusion

If you plan to run one source file in more than one environment then you may want to some pieces of code to be included or not included depending on the environment. The C language uses "#ifdef" and "#ifndef" for this purpose, and Lisp uses "#+" and "#-". The environment that the sharp sign macro checks is the *(status features)* list which is initialized when the Lisp system is built and which may be altered by *(sstatus feature foo)* and *(sstatus nofeature bar)* The form of conditional inclusion is

#+when what

where *when* is either a symbol or an expression involving symbols and the functions *and*, *or*, and *not*. The meaning is that *what* will only be read in if *when* is true. A symbol in *when* is true only if it appears in the *(status features)* list.

```
; suppose we want to write a program which references a file
; and which can run at ucb, ucsd and cmu where the file naming conventions
; are different.
;
-> (defun howold (name)
     (terpr)
     (load #+(or ucb ucsd) "/usr/lib/lisp/ages.l"
           #+cmu "/usr/lisp/doc/ages.l")
     (patom name)
     (patom " is ")
     (print (cdr (assoc name agefile)))
     (patom "years old")
     (terpr))
```

The form

#-when what

is equivalent to

#+(not when) what

8.3.4.2. fixnum character equivalents

When working with fixnum equivalents of characters, it is often hard to remember the number corresponding to a character. The form

#/c

is equivalent to the fixnum representation of character c.

```
; a function which returns t if the user types y else it returns nil.
;
-> (defun yesorno nil
     (progn (ans)
            (setq ans (tyi))
            (cond ((equal ans #/y) t)
                  (t nil))))
```

8.3.4.3. read time evaluation

Occasionally you want to express a constant as a Lisp expression, yet you don't want to pay the penalty of evaluating this expression each time it is referenced. The form

#.expression

evaluates the expression at read time and returns its value.

```
; a function to test if any of bits 1 3 or 12 are set in a fixnum.
;
-> (defun testit (num)
     (cond ((zerop (boole 1 num #.(+ (lsh 1 1) (lsh 1 3) (lsh 1 12))))
            nil)
           (t t)))
```

8.4. fclosures

Fclosures are a type of functional object. The purpose is to remember the values of some variables between invocations of the functional object and to protect this data from being inadvertently overwritten by other Lisp functions. Fortran programs usually exhibit this behavior for their variables. (In fact, some versions of Fortran would require the variables to be in COMMON). Thus it is easy to write a linear congruent random number generator in Fortran, merely by keeping the seed as a variable in the function. It is much more risky to do so in Lisp, since any special variable you picked, might be used by some other function. Fclosures are an attempt to provide most of the same functionality as closures in Lisp Machine Lisp, to users of FRANZ LISP. Fclosures are related to closures in this way:

(fclosure '(a b) 'foo) <==>
    (let ((a a) (b b)) (closure '(a b) 'foo))

8.4.1. an example

```
% lisp
Franz Lisp, Opus 38.60
->(defun code (me count)
  (print (list 'in x))
  (setq x (+ 1 x))
  (cond ((greaterp count 1) (funcall me me (sub1 count))))
  (print (list 'out x)))
code
->(defun tester (object count)
  (funcall object object count) (terpri))
tester
->(setq x 0)
0
->(setq z (fclosure '(x) 'code))
fclosure[8]
-> (tester z 3)
(in 0)(in 1)(in 2)(out 3)(out 3)(out 3)
nil
->x
0
```

The function *fclosure* creates a new object that we will call an fclosure, (although it is actually a vector). The fclosure contains a functional object, and a set of symbols and values for the symbols. In the above example, the fclosure functional object is the function code. The set of symbols and values just contains the symbol 'x' and zero, the value of 'x' when the fclosure was created.

When an fclosure is funcall'ed:

1) The Lisp system lambda binds the symbols in the fclosure to their values in the fclosure.
2) It continues the funcall on the functional object of the fclosure.
3) Finally, it un-lambda binds the symbols in the fclosure and at the same time stores the current values of the symbols in the fclosure.

Notice that the fclosure is saving the value of the symbol 'x'. Each time a fclosure is created, new space is allocated for saving the values of the symbols. Thus if we execute fclosure again, over the same function, we can have two independent counters:

```
-> (setq zz (fclosure '(x) 'code))
fclosure[1]
-> (tester zz 2)
(in 0)(in 1)(out 2)(out 2)
-> (tester zz 2)
(in 2)(in 3)(out 4)(out 4)
-> (tester z 3)
(in 3)(in 4)(in 5)(out 6)(out 6)(out 6)
```

8.4.2. useful functions

Here are some quick some summaries of functions dealing with closures. They are more formally defined in §2.8.4. To recap, fclosures are made by *(fclosure 'l_vars 'g_funcobj)*. l_vars is a list of symbols (not containing nil), g_funcobj is any object that can be funcalled. (Objects which can be funcalled, include compiled Lisp functions, lambda expressions, symbols, foreign functions, etc.) In general, if you want a compiled function to be closed over a variable, you must declare the variable to be special within the function. Another example would be:

```
(fclosure '(a b) #'(lambda (x) (plus x a)))
```

Here, the #' construction will make the compiler compile the lambda expression.

There are times when you want to share variables between fclosures. This can be done if the fclosures are created at the same time using *fclosure-list*. The function *fclosure-alist* returns an assoc list giving the symbols and values in the fclosure. The predicate *fclosurep* returns t iff its argument is a fclosure. Other functions imported from Lisp Machine Lisp are *symeval-in-fclosure, let-fclosed,* and *set-in-fclosure*. Lastly, the function *fclosure-function* returns the function argument.

8.4.3. internal structure

Currently, closures are implemented as vectors, with property being the symbol fclosure. The functional object is the first entry. The remaining entries are structures which point to the symbols and values for the closure, (with a reference count to determine if a recursive closure is active).

8.5. foreign subroutines and functions

FRANZ LISP has the ability to dynamically load object files produced by other compilers and to call functions defined in those files. These functions are called *foreign* functions.* There are seven types of foreign functions. They are characterized by the type of result they return, and by differences in the interpretation of their arguments. They come from two families: a group suited for languages which pass arguments by reference (e.g. Fortran), and a group suited for languages which pass arguments by value (e.g. C).

There are four types in the first group:

subroutine
: This does not return anything. The Lisp system always returns t after calling a subroutine.

function
: This returns whatever the function returns. This must be a valid Lisp object or it may cause the Lisp system to fail.

---

*This topic is also discussed in Report PAM-124 of the Center for Pure and Applied Mathematics, UCB, entitled "Parlez-Vous Franz? An Informal Introduction to Interfacing Foreign Functions to Franz LISP", by James R. Larus

integer-function
This returns an integer which the Lisp system makes into a fixnum and returns.

real-function
This returns a double precision real number which the Lisp system makes into a flonum and returns.

There are three types in the second group:

c-function
This is like an integer function, except for its different interpretation of arguments.

double-c-function
This is like a real-function.

vector-c-function
This is for C functions which return a structure. The first argument to such functions must be a vector (of type vectori), into which the result is stored. The second Lisp argument becomes the first argument to the C function, and so on

A foreign function is accessed through a binary object just like a compiled Lisp function. The difference is that the discipline field of a binary object for a foreign function is a string whose first character is given in the following table:

letter	type
s	subroutine
f	function
i	integer-function
r	real-function.
c	c-function
v	vector-c-function
d	double-c-function

Two functions are provided for setting-up foreign functions. *Cfasl* loads an object file into the Lisp system and sets up one foreign function binary object. If there are more than one function in an object file, *getaddress* can be used to set up additional foreign function objects.

Foreign functions are called just like other functions, e.g *(funname arg1 arg2)*. When a function in the Fortran group is called, the arguments are evaluated and then examined. List, hunk and symbol arguments are passed unchanged to the foreign function. Fixnum and flonum arguments are copied into a temporary location and a pointer to the value is passed (this is because Fortran uses call by reference and it is dangerous to modify the contents of a fixnum or flonum which something else might point to). If the argument is an array object, the data field of the array object is passed to the foreign function (This is the easiest way to send large amounts of data to and receive large amounts of data from a foreign function). If a binary object is an argument, the entry field of that object is passed to the foreign function (the entry field is the address of a function, so this amounts to passing a function as an argument).

When a function in the C group is called, fixnum and flownum arguments are passed by value. For almost all other arguments, the address is merely provided to the C routine. The only exception arises when you want to invoke a C routine which expects a "structure" argument. Recall that a (rarely used) feature of the C language is

the ability to pass structures by value. This copies the structure onto the stack. Since the Franz's nearest equivalent to a C structure is a vector, we provide an escape clause to copy the contents of an immediate-type vector by value. If the property field of a vectori argument, is the symbol "value-structure-argument", then the binary data of this immediate-type vector is copied into the argument list of the C routine.

The method a foreign function uses to access the arguments provided by Lisp is dependent on the language of the foreign function. The following scripts demonstrate how how Lisp can interact with three languages: C, Pascal and Fortran. C and Pascal have pointer types and the first script shows how to use pointers to extract information from Lisp objects. There are two functions defined for each language. The first (cfoo in C, pfoo in Pascal) is given four arguments, a fixnum, a flonum-block array, a hunk of at least two fixnums and a list of at least two fixnums. To demonstrate that the values were passed, each ?foo function prints its arguments (or parts of them). The ?foo function then modifies the second element of the flonum-block array and returns a 3 to Lisp. The second function (cmemq in C, pmemq in Pascal) acts just like the Lisp *memq* function (except it won't work for fixnums whereas the lisp *memq* will work for small fixnums). In the script, typed input is in **bold**, computer output is in roman and comments are in *italic.*

These are the C coded functions
% cat ch8auxc.c

```
/* demonstration of c coded foreign integer-function */

/* the following will be used to extract fixnums out of a list of fixnums */
struct listoffixnumscell
{    struct listoffixnumscell *cdr;
     int *fixnum;
};

struct listcell
{           struct listcell *cdr;
            int car;
};

cfoo(a,b,c,d)
int *a;
double b[];
int *c[];
struct listoffixnumscell *d;
{
   printf("a: %d, b[0]: %f, b[1]: %f0, *a, b[0], b[1]);
   printf(" c (first): %d   c (second): %d0,
                  *c[0],*c[1]);
   printf(" ( %d %d ... ) ", *(d->fixnum), *(d->cdr->fixnum));
   b[1] = 3.1415926;
   return(3);
}

struct listcell *
cmemq(element,list)
int element;
struct listcell *list;
{
  for( ; list && element != list->car ;  list = list->cdr);
  return(list);
}
```

These are the Pascal coded functions
% cat ch8auxp.p

```
type    pinteger = ^integer;
        realarray = array[0..10] of real;
        pintarray = array[0..10] of pinteger;
        listoffixnumscell = record
                                cdr  : ^listoffixnumscell;
                                fixnum : pinteger;
                            end;
        plistcell = ^listcell;
        listcell = record
                       cdr : plistcell;
                       car : integer;
                   end;

function pfoo ( var a : integer ;
                var b : realarray;
                var c : pintarray;
                var d : listoffixnumscell) : integer;
begin
  writeln(' a:',a, ' b[0]:', b[0], ' b[1]:', b[1]);
  writeln(' c (first):', c[0]^,' c (second):', c[1]^);
  writeln(' ( ', d.fixnum^, d.cdr^.fixnum^, ' ...) ');
  b[1] := 3.1415926;
  pfoo := 3
end ;

{ the function pmemq looks for the Lisp pointer given as the first argument
  in the list pointed to by the second argument.
  Note that we declare " a : integer " instead of " var a : integer " since
  we are interested in the pointer value instead of what it points to (which
  could be any Lisp object)
}
function pmemq( a : integer; list : plistcell) : plistcell;
begin
 while (list <> nil) and (list^.car <> a) do list := list^.cdr;
 pmemq := list;
end ;
```

The files are compiled
% cc -c ch8auxc.c
1.0u 1.2s 0:15 14% 30+39k 33+20io 147pf+0w
% pc -c ch8auxp.p
3.0u 1.7s 0:37 12% 27+32k 53+32io 143pf+0w

% lisp
Franz Lisp, Opus 38.60
First the files are loaded and we set up one foreign function binary. We have two functions in each file so we must choose one to tell cfasl about. The choice is arbitrary.
-> (cfasl 'ch8auxc.o '_cfoo 'cfoo "integer-function")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxc.o -e _cfoo -o /tmp/Li7055.0 -lc
#63000-"integer-function"
-> (cfasl 'ch8auxp.o '_pfoo 'pfoo "integer-function" "-lpc")
/usr/lib/lisp/nld -N -A /tmp/Li7055.0 -T 63200 ch8auxp.o -e _pfoo -o /tmp/Li7055.1 -lpc -lc
#63200-"integer-function"
Here we set up the other foreign function binary objects
-> (getaddress '_cmemq 'cmemq "function" '_pmemq 'pmemq "function")
#6306c-"function"
We want to create and initialize an array to pass to the cfoo function. In this case we create an unnamed array and store it in the value cell of testarr. When we create an array to pass to the Pascal program we will use a named array just to demonstrate the different way that named and unnamed arrays are created and accessed.
-> (setq testarr (array nil flonum-block 2))
array[2]
-> (store (funcall testarr 0) 1.234)

```
1.234
-> (store (funcall testarr 1) 5.678)
5.678
-> (cfoo 385 testarr (hunk 10 11 13 14) '(15 16 17))
a: 385, b[0]: 1.234000, b[1]: 5.678000
 c (first): 10   c (second): 11
 ( 15 16 ... )
 3
```

Note that cfoo has returned 3 as it should. It also had the side effect of changing the second value of the array to 3.1415926 which check next.

```
-> (funcall testarr 1)
3.1415926
```

In preparation for calling pfoo we create an array.

```
-> (array test flonum-block 2)
array[2]
-> (store (test 0) 1.234)
1.234
-> (store (test 1) 5.678)
5.678
-> (pfoo 385 (getd 'test) (hunk 10 11 13 14) '(15 16 17))
 a:      385 b[0]:  1.23400000000000E+00 b[1]:  5.67800000000000E+00
 c (first):       10 c (second):       11
 (       15       16 ...)
3
-> (test 1)
3.1415926
```

Now to test out the memq's

```
-> (cmemq 'a '(b c a d e f))
(a d e f)
-> (pmemq 'e '(a d f g a x))
nil
```

The Fortran example will be much shorter since in Fortran you can't follow pointers as you can in other languages. The Fortran function ffoo is given three arguments: a fixnum, a fixnum-block array and a flonum. These arguments are printed out to verify that they made it and then the first value of the array is modified. The function returns a double precision value which is converted to a flonum by lisp and printed. Note that the entry point corresponding to the Fortran function ffoo is _ffoo_ as opposed to the C and Pascal convention of preceding the name with an underscore.

```
% cat ch8auxf.f
          double precision function ffoo(a,b,c)
          integer a,b(10)
          double precision c
          print 2,a,b(1),b(2),c
2         format(' a=',i4,', b(1)=',i5,', b(2)=',i5,' c=',f6.4)
          b(1) = 22
          ffoo = 1.23456
          return
          end
% f77 -c ch8auxf.f
ch8auxf.f:
  ffoo:
```

```
0.9u 1.8s 0:12 22% 20+22k 54+48io 158pf+0w
% lisp
Franz Lisp, Opus 38.60
-> (cfasl 'ch8auxf.o '_ffoo_ 'ffoo "real-function" "-lF77 -lI77")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxf.o -e _ffoo_
-o /tmp/Li11066.0 -lF77 -lI77 -lc
#6307c-"real-function"

-> (array test fixnum-block 2)
array[2]
-> (store (test 0) 10)
10
-> (store (test 1) 11)
11
-> (ffoo 385 (getd 'test) 5.678)
 a= 385, b(1)=   10, b(2)=   11 c=5.6780
1.234559893608093
-> (test 0)
22
```

CHAPTER 9

Arrays and Vectors

Arrays and vectors are two means of expressing aggregate data objects in FRANZ LISP. Vectors may be thought of as sequences of data. They are intended as a vehicle for user-defined data types. This use of vectors is still experimental and subject to revision. As a simple data structure, they are similar to hunks and strings. Vectors are used to implement closures, and are useful to communicate with foreign functions. Both of these topics were discussed in Chapter 8. Later in this chapter, we describe the current implementation of vectors, and will advise the user what is most likely to change.

Arrays in FRANZ LISP provide a programmable data structure access mechanism. One possible use for FRANZ LISP arrays is to implement Maclisp style arrays which are simple vectors of fixnums, flonums or general lisp values. This is described in more detail in §9.3 but first we will describe how array references are handled by the lisp system.

The structure of an array object is given in §1.3.10 and reproduced here for your convenience.

Subpart name	Get value	Set value	Type
access function	getaccess	putaccess	binary, list or symbol
auxiliary	getaux	putaux	lispval
data	arrayref	replace set	block of contiguous lispval
length	getlength	putlength	fixnum
delta	getdelta	putdelta	fixnum

9.1. general arrays Suppose the evaluator is told to evaluate *(foo a b)* and the function cell of the symbol foo contains an array object (which we will call foo_arr_obj). First the evaluator will evaluate and stack the values of *a* and *b*. Next it will stack the array object foo_arr_obj. Finally it will call the access function of foo_arr_obj. The access function should be a lexpr[†] or a symbol whose function cell contains a lexpr. The access function is responsible for locating and returning a value from the array. The array access function is free to interpret the arguments as it wishes. The Maclisp compatible array access function which is provided in the standard FRANZ LISP system interprets the arguments as subscripts in the same way as languages like Fortran and Pascal.

[†]A lexpr is a function which accepts any number of arguments which are evaluated before the function is called.

The array access function will also be called upon to store elements in the array. For example, *(store (foo a b) c)* will automatically expand to (foo c a b) and when the evaluator is called to evaluate this, it will evaluate the arguments *c*, *b* and *a*. Then it will stack the array object (which is stored in the function cell of foo) and call the array access function with (now) four arguments. The array access function must be able to tell this is a store operation, which it can do by checking the number of arguments it has been given (a lexpr can do this very easily).

9.2. subparts of an array object An array is created by allocating an array object with *marray* and filling in the fields. Certain lisp functions interpret the values of the subparts of the array object in special ways as described in the following text. Placing illegal values in these subparts may cause the lisp system to fail.

9.2.1. access function The purpose of the access function has been described above. The contents of the access function should be a lexpr, either a binary (compiled function) or a list (interpreted function). It may also be a symbol whose function cell contains a function definition. This subpart is used by *eval*, *funcall*, and *apply* when evaluating array references.

9.2.2. auxiliary This can be used for any purpose. If it is a list and the first element of that list is the symbol unmarked_array then the data subpart will not be marked by the garbage collector (this is used in the Maclisp compatible array package and has the potential for causing strange errors if used incorrectly).

9.2.3. data This is either nil or points to a block of data space allocated by *segment* or *small-segment.*

9.2.4. length This is a fixnum whose value is the number of elements in the data block. This is used by the garbage collector and by *arrayref* to determine if your index is in bounds.

9.2.5. delta This is a fixnum whose value is the number of bytes in each element of the data block. This will be four for an array of fixnums or value cells, and eight for an array of flonums. This is used by the garbage collector and *arrayref* as well.

9.3. The Maclisp compatible array package

A Maclisp style array is similar to what is known as arrays in other languages: a block of homogeneous data elements which is indexed by one or more integers called subscripts. The data elements can be all fixnums, flonums or general lisp objects. An

array is created by a call to the function *array* or **array*. The only difference is that **array* evaluates its arguments. This call: *(array foo t 3 5)* sets up an array called foo of dimensions 3 by 5. The subscripts are zero based. The first element is *(foo 0 0)*, the next is *(foo 0 1)* and so on up to *(foo 2 4)*. The t indicates a general lisp object array which means each element of foo can be any type. Each element can be any type since all that is stored in the array is a pointer to a lisp object, not the object itself. *Array* does this by allocating an array object with *marray* and then allocating a segment of 15 consecutive value cells with *small-segment* and storing a pointer to that segment in the data subpart of the array object. The length and delta subpart of the array object are filled in (with 15 and 4 respectively) and the access function subpart is set to point to the appropriate array access function. In this case there is a special access function for two dimensional value cell arrays called arrac-twoD, and this access function is used. The auxiliary subpart is set to (t 3 5) which describes the type of array and the bounds of the subscripts. Finally this array object is placed in the function cell of the symbol foo. Now when *(foo 1 3)* is evaluated, the array access function is invoked with three arguments: 1, 3 and the array object. From the auxiliary field of the array object it gets a description of the particular array. It then determines which element *(foo 1 3)* refers to and uses arrayref to extract that element. Since this is an array of value cells, what arrayref returns is a value cell whose value is what we want, so we evaluate the value cell and return it as the value of *(foo 1 3)*.

In Maclisp the call *(array foo fixnum 25)* returns an array whose data object is a block of 25 memory words. When fixnums are stored in this array, the actual numbers are stored instead of pointers to the numbers as is done in general lisp object arrays. This is efficient under Maclisp but inefficient in FRANZ LISP since every time a value was referenced from an array it had to be copied and a pointer to the copy returned to prevent aliasing[†]. Thus t, fixnum and flonum arrays are all implemented in the same manner. This should not affect the compatibility of Maclisp and FRANZ LISP. If there is an application where a block of fixnums or flonums is required, then the exact same effect of fixnum and flonum arrays in Maclisp can be achieved by using fixnum-block and flonum-block arrays. Such arrays are required if you want to pass a large number of arguments to a Fortran or C coded function and then get answers back.

The Maclisp compatible array package is just one example of how a general array scheme can be implemented. Another type of array you could implement would be hashed arrays. The subscript could be anything, not just a number. The access function would hash the subscript and use the result to select an array element. With the generality of arrays also comes extra cost; if you just want a simple aggregate of (less than 128) general lisp objects you would be wise to look into using hunks.

9.4. vectors Vectors were invented to fix two shortcommings with hunks. They can be longer than 128 elements. They also have a tag associated with them, which is intended to say, for example, "Think of me as an *Blobit.*" Thus a **vector** is an arbitrary sized hunk with a property list.

Continuing the example, the lisp kernel may not know how to print out or evaluate *blobits*, but this is information which will be common to all *blobits*. On the other

[†]Aliasing is when two variables are share the same storage location. For example if the copying mentioned weren't done then after *(setq x (foo 2))* was done, the value of x and (foo 2) would share the same location. Then should the value of (foo 2) change, x's value would change as well. This is considered dangerous and as a result pointers are never returned into the data space of arrays.

hand, for each individual blobits there are particulars which are likely to change, (height, weight, eye-color). This is the part that would previously have been stored in the individual entries in the hunk, and are stored in the data slots of the vector. Once again we summarize the structure of a vector in tabular form:

Subpart name	Get value	Set value	Type
datum[*i*]	vref	vset	lispval
property	vprop	vsetprop vputprop	lispval
size	vsize	-	fixnum

Vectors are created specifying size and optional fill value using the function (*new-vector* 'x_size ['g_fill ['g_prop]]), or by initial values: (*vector* ['g_val ...]).

9.5. anatomy of vectors There are some technical details about vectors, that the user should know:

9.5.1. size The user is not free to alter this. It is noted when the vector is created, and is used by the garbage collector. The garbage collector will coallesce two free vectors, which are neighbors in the heap. Internally, this is kept as the number of bytes of data. Thus, a vector created by (*vector* 'foo), has a size of 4.

9.5.2. property Currently, we expect the property to be either a symbol, or a list whose first entry is a symbol. The symbols **fclosure** and **structure-value-argument** are magic, and their effect is described in Chapter 8. If the property is a (non-null) symbol, the vector will be printed out as <symbol>[<size>]. Another case is if the property is actually a (disembodied) property-list, which contains a value for the indicator **print**. The value is taken to be a Lisp function, which the printer will invoke with two arguments: the vector and the current output port. Otherwise, the vector will be printed as vector[<size>]. We have vague (as yet unimplemented) ideas about similar mechanisms for evaluation properties. Users are cautioned against putting anything other than nil in the property entry of a vector.

9.5.3. internal order In memory, vectors start with a longword containing the size (which is immediate data within the vector). The next cell contains a pointer to the property. Any remaining cells (if any) are for data. Vectors are handled differently from any other object in FRANZ LISP, in that a pointer to a vector is pointer to the first data cell, i.e. a pointer to the *third* longword of the structure. This was done for efficiency in compiled code and for uniformity in referencing immediate-vectors (described below). The user should never return a pointer to any other part of a vector, as this may cause the garbage collector to follow an invalid pointer.

9.6. immediate-vectors Immediate-vectors are similar to vectors. They differ, in that binary data are stored in space directly within the vector. Thus the garbage collector will preserve the vector itself (if used), and will only traverse the property cell. The data may be referenced as longwords, shortwords, or even bytes. Shorts and bytes are returned sign-extended. The compiler open-codes such references, and will avoid boxing the resulting integer data, where possible. Thus, immediate vectors may be used for efficiently processing character data. They are also useful in storing results from functions written in other languages.

Subpart name	Get value	Set value	Type
datum[*i*]	vrefi-byte vrefi-word vrefi-long	vseti-byte vseti-word vseti-long	fixnum fixnum fixnum
property	vprop	vsetprop vputprop	lispval
size	vsize vsize-byte vsize-word	-	fixnum fixnum fixnum

To create immediate vectors specifying size and fill data, you can use the functions *new-vectori-byte*, *new-vectori-word*, or *new-vectori-long*. You can also use the functions *vectori-byte*, *vectori-word*, or *vectori-long*. All of these functions are described in chapter 2.

CHAPTER 10

Exception Handling

10.1. Errset and Error Handler Functions

FRANZ LISP allows the user to handle in a number of ways the errors which arise during computation. One way is through the use of the *errset* function. If an error occurs during the evaluation of the *errset*'s first argument, then the locus of control will return to the errset which will return nil (except in special cases, such as *err*). The other method of error handling is through an error handler function. When an error occurs, the error handler is called and is given as an argument a description of the error which just occurred. The error handler may take one of the following actions:

(1) it could take some drastic action like a *reset* or a *throw*.

(2) it could, assuming that the error is continuable, return to the function which noticed the error. The error handler indicates that it wants to return a value from the error by returning a list whose *car* is the value it wants to return.

(3) it could decide not to handle the error and return a non-list to indicate this fact.

10.2. The Anatomy of an error

Each error is described by a list of these items:

(1) error type - This is a symbol which indicates the general classification of the error. This classification may determine which function handles this error.

(2) unique id - This is a fixnum unique to this error.

(3) continuable - If this is non-nil then this error is continuable. There are some who feel that every error should be continuable and the reason that some (in fact most) errors in FRANZ LISP are not continuable is due to the laziness of the programmers.

(4) message string - This is a symbol whose print name is a message describing the error.

(5) data - There may be from zero to three lisp values which help describe this particular error. For example, the unbound variable error contains one datum value, the symbol whose value is unbound. The list describing that error might look like:

(ER%misc 0 t |Unbound Variable:| foobar)

10.3. Error handling algorithm

This is the sequence of operations which is done when an error occurs:

(1) If the symbol **ER%all** has a non nil value then this value is the name of an error handler function. That function is called with a description of the error. If that function returns (and of course it may choose not to) and the value is a list and this error is continuable, then we return the *car* of the list to the function which called the error. Presumably the function will use this value to retry the operation. On the other hand, if the error handler returns a non list, then it has chosen not to handle this error, so we go on to step (2). Something special happens before we call the **ER%all** error handler which does not happen in any of the other cases we will describe below. To help insure that we don't get infinitely recursive errors if **ER%all** is set to a bad value, the value of **ER%all** is set to nil before the handler is called. Thus it is the responsibility of the **ER%all** handler to 'reenable' itself by storing its name in **ER%all.**

(2) Next the specific error handler for the type of error which just occurred is called (if one exists) to see if it wants to handle the error. The names of the handlers for the specific types of errors are stored as the values of the symbols whose names are the types. For example the handler for miscellaneous errors is stored as the value of **ER%misc.** Of course, if **ER%misc** has a value of nil, then there is no error handler for this type of error. Appendix B contains list of all error types. The process of classifying the errors is not complete and thus most errors are lumped into the **ER%misc** category. Just as in step (1), the error handler function may choose not to handle the error by returning a non-list, and then we go to step (3).

(3) Next a check is made to see if there is an *errset* surrounding this error. If so the second argument to the *errset* call is examined. If the second argument was not given or is non nil then the error message associated with this error is printed. Finally the stack is popped to the context of the *errset* and then the *errset* returns nil. If there was no *errset* we go to step (4).

(4) If the symbol **ER%tpl** has a value then it is the name of an error handler which is called in a manner similar to that discussed above. If it chooses not to handle the error, we go to step (5).

(5) At this point it has been determined that the user doesn't want to handle this error. Thus the error message is printed out and a *reset* is done to send the flow of control to the top-level.

To summarize the error handling system: When an error occurs, you have two chances to handle it before the search for an *errset* is done. Then, if there is no *errset*, you have one more chance to handle the error before control jumps to the top level. Every error handler works in the same way: It is given a description of the error (as described in the previous section). It may or may not return. If it returns, then it returns either a list or a non-list. If it returns a list and the error is continuable, then the *car* of the list is returned to the function which noticed the error. Otherwise the error handler has decided not to handle the error and we go on to something else.

10.4. Default aids

There are two standard error handlers which will probably handle the needs of most users. One of these is the lisp coded function *break-err-handler* which is the default value of **ER%tpl.** Thus when all other handlers have ignored an error, *break-*

err-handler will take over. It will print out the error message and go into a read-eval-print loop. The other standard error handler is *debug-err-handler*. This handler is designed to be connected to **ER%all**and is useful if your program uses *errset* and you want to look at the error before it is thrown up to the *errset*.

10.5. Autoloading

When *eval*, *apply* or *funcall* are told to call an undefined function, an **ER%undef** error is signaled. The default handler for this error is *undef-func-handler*. This function checks the property list of the undefined function for the indicator autoload. If present, the value of that indicator should be the name of the file which contains the definition of the undefined function. *Undef-func-handler* will load the file and check if it has defined the function which caused the error. If it has, the error handler will return and the computation will continue as if the error did not occur. This provides a way for the user to tell the lisp system about the location of commonly used functions. The trace package sets up an autoload property to point to /usr/lib/lisp/trace.

10.6. Interrupt processing

The UNIX operating system provides one user interrupt character which defaults to ^C.† The user may select a lisp function to run when an interrupt occurs. Since this interrupt could occur at any time, and in particular could occur at a time when the internal stack pointers were in an inconsistent state, the processing of the interrupt may be delayed until a safe time. When the first ^C is typed, the lisp system sets a flag that an interrupt has been requested. This flag is checked at safe places within the interpreter and in the *qlinker* function. If the lisp system doesn't respond to the first ^C, another ^C should be typed. This will cause all of the transfer tables to be cleared forcing all calls from compiled code to go through the *qlinker* function where the interrupt flag will be checked. If the lisp system still doesn't respond, a third ^C will cause an immediate interrupt. This interrupt will not necessarily be in a safe place so the user should *reset* the lisp system as soon as possible.

†Actually there are two but the lisp system does not allow you to catch the QUIT interrupt.

CHAPTER 11

The Joseph Lister Trace Package

The Joseph Lister† Trace package is an important tool for the interactive debugging of a Lisp program. It allows you to examine selected calls to a function or functions, and optionally to stop execution of the Lisp program to examine the values of variables.

The trace package is a set of Lisp programs located in the Lisp program library (usually in the file /usr/lib/lisp/trace.l). Although not normally loaded in the Lisp system, the package will be loaded in when the first call to *trace* is made.

(trace [ls_arg1 ...])

WHERE: the form of the ls_arg*i* is described below.

RETURNS: a list of the function sucessfully modified for tracing. If no arguments are given to *trace*, a list of all functions currently being traced is returned.

SIDE EFFECT: The function definitions of the functions to trace are modified.

The ls_arg*i* can have one of the following forms:

foo - when foo is entered and exited, the trace information will be printed.

(foo break) - when foo is entered and exited the trace information will be printed. Also, just after the trace information for foo is printed upon entry, you will be put in a special break loop. The prompt is 'T>' and you may type any Lisp expression, and see its value printed. The *i*th argument to the function just called can be accessed as (arg *i*). To leave the trace loop, just type ^D or (tracereturn) and execution will continue. Note that ^D will work only on UNIX systems.

(foo if expression) - when foo is entered and the expression evaluates to non-nil, then the trace information will be printed for both exit and entry. If expression evaluates to nil, then no trace information will be printed.

(foo ifnot expression) - when foo is entered and the expression evaluates to nil, then the trace information will be printed for both entry and exit. If both **if** and **ifnot** are specified, then the **if** expression must evaluate to non nil AND the **ifnot** expression must evaluate to nil for the trace information to be printed out.

---

†*Lister, Joseph* 1st Baron Lister of Lyme Regis, 1827-1912; English surgeon: introduced antiseptic surgery.

(foo evalin expression) - when foo is entered and after the entry trace information is printed, expression will be evaluated. Exit trace information will be printed when foo exits.

(foo evalout expression) - when foo is entered, entry trace information will be printed. When foo exits, and before the exit trace information is printed, expression will be evaluated.

(foo evalinout expression) - this has the same effect as (trace (foo evalin expression evalout expression)).

(foo lprint) - this tells *trace* to use the level printer when printing the arguments to and the result of a call to foo. The level printer prints only the top levels of list structure. Any structure below three levels is printed as a &. This allows you to trace functions with massive arguments or results.

The following trace options permit one to have greater control over each action which takes place when a function is traced. These options are only meant to be used by people who need special hooks into the trace package. Most people should skip reading this section.

(foo traceenter tefunc) - this tells *trace* that the function to be called when foo is entered is tefunc. tefunc should be a lambda of two arguments, the first argument will be bound to the name of the function being traced, foo in this case. The second argument will be bound to the list of arguments to which foo should be applied. The function tefunc should print some sort of "entering foo" message. It should not apply foo to the arguments, however. That is done later on.

(foo traceexit txfunc) - this tells *trace* that the function to be called when foo is exited is txfunc. txfunc should be a lambda of two arguments, the first argument will be bound to the name of the function being traced, foo in this case. The second argument will be bound to the result of the call to foo. The function txfunc should print some sort of "exiting foo" message.

(foo evfcn evfunc) - this tells *trace* that the form evfunc should be evaluated to get the value of foo applied to its arguments. This option is a bit different from the other special options since evfunc will usually be an expression, not just the name of a function, and that expression will be specific to the evaluation of function foo. The argument list to be applied will be available as T-arglist.

(foo printargs prfunc) - this tells *trace* to used prfunc to print the arguments to be applied to the function foo. prfunc should be a lambda of one argument. You might want to use this option if you wanted a print function which could handle circular lists. This option will work only if you do not specify your own **traceenter** function. Specifying the option **lprint** is just a simple way of changing the printargs function to the level printer.

(foo printres prfunc) - this tells *trace* to use prfunc to print the result of evaluating foo. prfunc should be a lambda of one argument. This option will work only if you do not specify your own **traceexit** function. Specifying the option **lprint** changes printres to the level printer.

You may specify more than one option for each function traced. For example:

(trace (foo if (eq 3 (arg 1)) break lprint) (bar evalin (print xyzzy)))

This tells *trace* to trace two more functions, foo and bar. Should foo be called with the first argument *eq* to 3, then the entering foo message will be printed with the level printer. Next it will enter a trace break loop, allowing you to evaluate any lisp expressions. When you exit the trace break loop, foo will be applied to its arguments and the resulting value will be printed, again using the level printer. Bar is also traced, and each time bar is entered, an entering bar message will be printed and then the value of xyzzy will be printed. Next bar will be applied to its arguments and the result will be printed. If you tell *trace* to trace a function which is already traced, it will first *untrace* it. Thus if you want to specify more than one trace option for a function, you must do it all at once. The following is *not* equivalent to the preceding call to *trace* for foo:

(trace (foo if (eq 3 (arg 1))) (foo break) (foo lprint))

In this example, only the last option, lprint, will be in effect.

If the symbol $tracemute is given a non nil value, printing of the function name and arguments on entry and exit will be surpressed. This is particularly useful if the function you are tracing fails after many calls to it. In this case you would tell *trace* to trace the function, set $tracemute to t, and begin the computation. When an error occurs you can use *tracedump* to print out the current trace frames.

Generally the trace package has its own internal names for the the lisp functions it uses, so that you can feel free to trace system functions like *cond* and not worry about adverse interaction with the actions of the trace package. You can trace any type of function: lambda, nlambda, lexpr or macro whether compiled or interpreted and you can even trace array references (however you should not attempt to store in an array which has been traced).

When tracing compiled code keep in mind that many function calls are translated directly to machine language or other equivalent function calls. A full list of open coded functions is listed at the beginning of the liszt compiler source. *Trace* will do a *(sstatus translink nil)* to insure that the new traced definitions it defines are called instead of the old untraced ones. You may notice that compiled code will run slower after this is done.

(traceargs s_func [x_level])

WHERE: if x_level is missing it is assumed to be 1.

RETURNS: the arguments to the x_level*th* call to traced function s_func are returned.

(tracedump)

SIDE EFFECT: the currently active trace frames are printed on the terminal. returns a list of functions untraced.

(untrace [s_arg1 ...])

RETURNS: a list of the functions which were untraced.

NOTE: if no arguments are given, all functions are untraced.

SIDE EFFECT: the old function definitions of all traced functions are restored except in the case where it appears that the current definition of a function was not created by trace.

CHAPTER 12

Liszt - the lisp compiler

12.1. General strategy of the compiler

The purpose of the lisp compiler, Liszt, is to create an object module which when brought into the lisp system using *fasl* will have the same effect as bringing in the corresponding lisp coded source module with *load* with one important exception, functions will be defined as sequences of machine language instructions, instead of lisp S-expressions. Liszt is not a function compiler, it is a *file* compiler. Such a file can contain more than function definitions; it can contain other lisp S-expressions which are evaluated at load time. These other S-expressions will also be stored in the object module produced by Liszt and will be evaluated at fasl time.

As is almost universally true of Lisp compilers, the main pass of Liszt is written in Lisp. A subsequent pass is the assembler, for which we use the standard UNIX assembler.

12.2. Running the compiler

The compiler is normally run in this manner:

% liszt foo

will compile the file foo.l or foo (the preferred way to indicate a lisp source file is to end the file name with '.l'). The result of the compilation will be placed in the file foo.o if no fatal errors were detected. All messages which Liszt generates go to the standard output. Normally each function name is printed before it is compiled (the –q option suppresses this).

12.3. Special forms

Liszt makes one pass over the source file. It processes each form in this way:

12.3.1. macro expansion

If the form is a macro invocation (i.e it is a list whose car is a symbol whose function binding is a macro), then that macro invocation is expanded. This is repeated until the top level form is not a macro invocation. When Liszt begins, there are already some macros defined, in fact some functions (such as defun) are actually macros. The user may define his own macros as well. For a macro to be used it must be defined in the Lisp system in which Liszt runs.

12.3.2. classification

After all macro expansion is done, the form is classified according to its *car* (if the form is not a list, then it is classified as an *other*).

12.3.2.1. eval-when

The form of eval-when is *(eval-when (time1 time2 ...) form1 form2 ...)* where the time*i* are one of *eval*, *compile*, or *load*. The compiler examines the form*i* in sequence and the action taken depends on what is in the time list. If *compile* is in the list then the compiler will invoke *eval* on each form*i* as it examines it. If *load* is in the list then the compile will recursively call itself to compile each form*i* as it examines it. Note that if *compile* and *load* are in the time list, then the compiler will both evaluate and compile each form. This is useful if you need a function to be defined in the compiler at both compile time (perhaps to aid macro expansion) and at run time (after the file is *fasl*ed in).

12.3.2.2. declare

Declare is used to provide information about functions and variables to the compiler. It is (almost) equivalent to *(eval-when (compile) ...)*. You may declare functions to be one of three types: lambda (*expr), nlambda (*fexpr), lexpr (*lexpr). The names in parenthesis are the Maclisp names and are accepted by the compiler as well (and not just when the compiler is in Maclisp mode). Functions are assumed to be lambdas until they are declared otherwise or are defined differently. The compiler treats calls to lambdas and lexprs equivalently, so you needn't worry about declaring lexprs either. It is important to declare nlambdas or define them before calling them. Another attribute you can declare for a function is localf which makes the function 'local'. A local function's name is known only to the functions defined within the file itself. The advantage of a local function is that is can be entered and exited very quickly and it can have the same name as a function in another file and there will be no name conflict.

Variables may be declared special or unspecial. When a special variable is lambda bound (either in a lambda, prog or do expression), its old value is stored away on a stack for the duration of the lambda, prog or do expression. This takes time and is often not necessary. Therefore the default classification for variables is unspecial. Space for unspecial variables is dynamically allocated on a stack. An unspecial variable can only be accessed from within the function where it is created by its presence in a lambda, prog or do expression variable list. It is possible to declare that all variables are special as will be shown below.

You may declare any number of things in each declare statement. A sample declaration is

```
(declare
    (lambda func1 func2)
    (*fexpr func3)
    (*lexpr func4)
    (localf func5)
    (special var1 var2 var3)
    (unspecial var4))
```

You may also declare all variables to be special with *(declare (specials t))*. You may declare that macro definitions should be compiled as well as evaluated at compile time by *(declare (macros t))*. In fact, as was mentioned above, declare is much like *(eval-when (compile) ...)*. Thus if the compiler sees *(declare (foo bar))* and foo is defined, then it will evaluate *(foo bar)*. If foo is not defined then an undefined declare attribute warning will be issued.

12.3.2.3. (progn 'compile form1 form2 ... formn)

When the compiler sees this it simply compiles form1 through formn as if they too were seen at top level. One use for this is to allow a macro at top-level to expand into more than one function definition for the compiler to compile.

12.3.2.4. include/includef

Include and *includef* cause another file to be read and compiled by the compiler. The result is the same as if the included file were textually inserted into the original file. The only difference between *include* and *includef* is that include doesn't evaluate its argument and includef does. Nested includes are allowed.

12.3.2.5. def

A def form is used to define a function. The macros *defun* and *defmacro* expand to a def form. If the function being defined is a lambda, nlambda or lexpr then the compiler converts the lisp definition to a sequence of machine language instructions. If the function being defined is a macro, then the compiler will evaluate the definition, thus defining the macro withing the running Lisp compiler. Furthermore, if the variable *macros* is set to a non nil value, then the macro definition will also be translated to machine language and thus will be defined when the object file is fasled in. The variable *macros* is set to t by *(declare (macros t))*.

When a function or macro definition is compiled, macro expansion is done whenever possible. If the compiler can determine that a form would be evaluated if this function were interpreted then it will macro expand it. It will not macro expand arguments to a nlambda unless the characteristics of the nlambda is known (as is the case with *cond)*. The map functions (*map*, *mapc*, *mapcar*, and so on) are expanded to a *do* statement. This allows the first argument to the map function to be a lambda expression which references local variables of the function being defined.

12.3.2.6. other forms

All other forms are simply stored in the object file and are evaluated when the file is *fasl*ed in.

12.4. Using the compiler

The previous section describes exactly what the compiler does with its input. Generally you won't have to worry about all that detail as files which work interpreted will work compiled. Following is a list of steps you should follow to insure that a file will compile correctly.

[1] Make sure all macro definitions precede their use in functions or other macro definitions. If you want the macros to be around when you *fasl* in the object file you should include this statement at the beginning of the file: *(declare (macros t))*

[2] Make sure all nlambdas are defined or declared before they are used. If the compiler comes across a call to a function which has not been defined in the current file, which does not currently have a function binding, and whose type has not been declared then it will assume that the function needs its arguments evaluated (i.e. it is a lambda or lexpr) and will generate code accordingly. This means that you do not have to declare nlambda functions like *status* since they have an nlambda function binding.

[3] Locate all variables which are used for communicating values between functions. These variables must be declared special at the beginning of a file. In most cases there won't be many special declarations but if you fail to declare a variable special that should be, the compiled code could fail in mysterious ways. Let's look at a common problem, assume that a file contains just these three lines:

```
(def aaa (lambda (glob loc) (bbb loc)))
(def bbb (lambda (myloc) (add glob myloc)))
(def ccc (lambda (glob loc) (bbb loc)))
```

We can see that if we load in these two definitions then (aaa 3 4) is the same as (add 3 4) and will give us 7. Suppose we compile the file containing these definitions. When Liszt compiles aaa, it will assume that both glob and loc are local variables and will allocate space on the temporary stack for their values when aaa is called. Thus the values of the local variables glob and loc will not affect the values of the symbols glob and loc in the Lisp system. Now Liszt moves on to function bbb. Myloc is assumed to be local. When it sees the add statement, it find a reference to a variable called glob. This variable is not a local variable to this function and therefore glob must refer to the value of the symbol glob. Liszt will automatically declare glob to be special and it will print a warning to that effect. Thus subsequent uses of glob will always refer to the symbol glob. Next Liszt compiles ccc and treats glob as a special and loc as a local. When the object file is *fasl*'ed in, and (ccc 3 4) is evaluated, the symbol glob will be lambda bound to 3 bbb will be called and will return 7. However (aaa 3 4) will fail since when bbb is called, glob will be unbound. What should be done here is to put *(declare (special glob)* at the beginning of the file.

[4] Make sure that all calls to *arg* are within the lexpr whose arguments they reference. If *foo* is a compiled lexpr and it calls *bar* then *bar* cannot use *arg* to get at *foo*'s arguments. If both *foo* and *bar* are interpreted this will work however. The macro *listify* can be used to put all of some of a lexprs arguments in a list which then can be passed to other functions.

12.5. Compiler options

The compiler recognizes a number of options which are described below. The options are typed anywhere on the command line preceded by a minus sign. The entire command line is scanned and all options recorded before any action is taken. Thus
% liszt -mx foo
% liszt -m -x foo
% liszt foo -mx
are all equivalent. Before scanning the command line for options, liszt looks for in the environment for the variable LISZT, and if found scans its value as if it was a string of options. The meaning of the options are:

C The assembler language output of the compiler is commented. This is useful when debugging the compiler and is not normally done since it slows down compilation.

I The next command line argument is taken as a filename, and loaded prior to compilation.

e Evaluate the next argument on the command line before starting compilation. For example
% liszt -e '(setq foobar "foo string")' foo
will evaluate the above s-expression. Note that the shell requires that the arguments be surrounded by single quotes.

i Compile this program in interlisp compatibility mode. This is not implemented yet.

m Compile this program in Maclisp mode. The reader syntax will be changed to the Maclisp syntax and a file of macro definitions will be loaded in (usually named /usr/lib/lisp/machacks). This switch brings us sufficiently close to Maclisp to allow us to compile Macsyma, a large Maclisp program. However Maclisp is a moving target and we can't guarantee that this switch will allow you to compile any given program.

o Select a different object or assembler language file name. For example
% liszt foo -o xxx.o
will compile foo and into xxx.o instead of the default foo.o, and
% liszt bar -S -o xxx.s
will compile to assembler language into xxx.s instead of bar.s.

p place profiling code at the beginning of each non-local function. If the lisp system is also created with profiling in it, this allows function calling frequency to be determined (see *prof(1)*)

q Run in quiet mode. The names of functions being compiled and various "Note"'s are not printed.

Q print compilation statistics and warn of strange constructs. This is the inverse of the **q** switch and is the default.

r place bootstrap code at the beginning of the object file, which when the object file is executed will cause a lisp system to be invoked and the object file *fasl*ed in. This is known as 'autorun' and is described below.

S Create an assembler language file only.
% liszt -S foo
will create the file assembler language file foo.s and will not attempt to assemble it. If this option is not specified, the assembler language file will be put in the temporary disk area under a automatically generated name based on the lisp

compiler's process id. Then if there are no compilation errors, the assembler will be invoked to assemble the file.

T Print the assembler language output on the standard output file. This is useful when debugging the compiler.

u Run in UCI-Lisp mode. The character syntax is changed to that of UCI-Lisp and a UCI-Lisp compatibility package of macros is read in.

w Suppress warning messages.

x Create an cross reference file.
% liszt -x foo
not only compiles foo into foo.o but also generates the file foo.x . The file foo.x is lisp readable and lists for each function all functions which that function could call. The program lxref reads one or more of these ".x" files and produces a human readable cross reference listing.

12.6. autorun

The object file which liszt writes does not contain all the functions necessary to run the lisp program which was compiled. In order to use the object file, a lisp system must be started and the object file *fasl*ed in. When the -r switch is given to liszt, the object file created will contain a small piece of bootstrap code at the beginning, and the object file will be made executable. Now, when the name of the object file is given to the UNIX command interpreter (shell) to run, the bootstrap code at the beginning of the object file will cause a lisp system to be started and the first action the lisp system will take is to *fasl* in the object file which started it. In effect the object file has created an environment in which it can run.

Autorun is an alternative to *dumplisp*. The advantage of autorun is that the object file which starts the whole process is typically small, whereas the minimum *dumplisp*ed file is very large (one half megabyte). The disadvantage of autorun is that the file must be *fasl*ed into a lisp each time it is used whereas the file which *dumplisp* creates can be run as is. liszt itself is a *dumplisp*ed file since it is used so often and is large enough that too much time would be wasted *fasl*ing it in each time it was used. The lisp cross reference program, lxref, uses *autorun* since it is a small and rarely used program.

In order to have the program *fasl*ed in begin execution (rather than starting a lisp top level), the value of the symbol user-top-level should be set to the name of the function to get control. An example of this is shown next.

---

we want to replace the unix date program with one written in lisp.

```
% cat lispdate.l
(defun mydate nil
   (patom "The date is ")
   (patom (status ctime))
   (terpr)
   (exit 0))
(setq user-top-level 'mydate)

% liszt -r lispdate
Compilation begins with Lisp Compiler 5.2
source: lispdate.l, result: lispdate.o
mydate
%Note: lispdate.l: Compilation complete
%Note: lispdate.l:  Time: Real: 0:3, CPU: 0:0.28, GC: 0:0.00 for 0 gcs
%Note: lispdate.l: Assembly begins
%Note: lispdate.l: Assembly completed successfully
3.0u 2.0s 0:17 29%

 We change the name to remove the ".o", (this isn't necessary)
% mv lispdate.o lispdate

 Now we test it out
% lispdate
The date is Sat Aug  1 16:58:33 1981
%
```

12.7. pure literals

Normally the quoted lisp objects (literals) which appear in functions are treated as constants. Consider this function:

```
(def foo
   (lambda nil (cond ((not (eq 'a (car (setq x '(a b)))))
                      (print 'impossible!!))
                     (t (rplaca x 'd)))))
```

At first glance it seems that the first cond clause will never be true, since the *car* of *(a b)* should always be *a*. However if you run this function twice, it will print 'impossible!!' the second time. This is because the following clause modifies the 'constant' list *(a b)* with the *rplaca* function. Such modification of literal lisp objects can cause programs to behave strangely as the above example shows, but more importantly it can cause garbage collection problems if done to compiled code. When a file is *fasl*ed in, if the symbol $purcopylits is non nil, the literal lisp data is put in 'pure' space, that is it put in space which needn't be looked at by the garabage collector. This reduces the work the garbage collector must do but it is dangerous since if the literals are modified to point to non pure objects, the marker may not mark the non pure objects. If the symbol $purcopylits is nil then the literal lisp data is put in impure space and the compiled code will act like the interpreted code when literal data is modified. The default value for $purcopylits is t.

12.8. transfer tables

A transfer table is setup by *fasl* when the object file is loaded in. There is one entry in the transfer table for each function which is called in that object file. The entry for a call to the function *foo* has two parts whose contents are:

[1] function address – This will initially point to the internal function *qlinker*. It may some time in the future point to the function *foo* if certain conditions are satisfied (more on this below).

[2] function name – This is a pointer to the symbol *foo*. This will be used by *qlinker*.

When a call is made to the function *foo* the call will actually be made to the address in the transfer table entry and will end up in the *qlinker* function. *Qlinker* will determine that *foo* was the function being called by locating the function name entry in the transfer table†. If the function being called is not compiled then *qlinker* just calls *funcall* to perform the function call. If *foo* is compiled and if *(status translink)* is non nil, then *qlinker* will modify the function address part of the transfer table to point directly to the function *foo*. Finally *qlinker* will call *foo* directly . The next time a call is made to *foo* the call will go directly to *foo* and not through *qlinker*. This will result in a substantial speedup in compiled code to compiled code transfers. A disadvantage is that no debugging information is left on the stack, so *showstack* and *baktrace* are useless. Another disadvantage is that if you redefine a compiled function either through loading in a new version or interactively defining it, then the old version may still be called from compiled code if the fast linking described above has already been done. The solution to these problems is to use *(sstatus translink value)*. If value is

nil All transfer tables will be cleared, i.e. all function addresses will be set to point to *qlinker*. This means that the next time a function is called *qlinker* will be called and will look at the current definition. Also, no fast links will be set up since *(status translink)* will be nil. The end result is that *showstack* and *baktrace* will work and the function definition at the time of call will always be used.

on This causes the lisp system to go through all transfer tables and set up fast links wherever possible. This is normally used after you have *fasl*ed in all of your files. Furthermore since *(status translink)* is not nil, *qlinker* will make new fast links if the situation arises (which isn't likely unless you *fasl* in another file).

t This or any other value not previously mentioned will just make *(status translink)* be non nil, and as a result fast links will be made by *qlinker* if the called function is compiled.

12.9. Fixnum functions

The compiler will generate inline arithmetic code for fixnum only functions. Such functions include +, –, *, /, \, 1+ and 1–. The code generated will be much faster than using *add*, *difference*, etc. However it will only work if the arguments to and results of the functions are fixnums. No type checking is done.

---

†*Qlinker* does this by tracing back the call stack until it finds the *calls* machine instruction which called it. The address field of the *calls* contains the address of the transfer table entry.

CHAPTER 13

The CMU User Toplevel and the File Package

This documentation was written by Don Cohen, and the functions described below were imported from PDP-10 CMULisp.

Non CMU users note: this is not the default top level for your Lisp system. In order to start up this top level, you should type *(load 'cmuenv)*.

13.1. User Command Input Top Level

The top-level is the function that reads what you type, evaluates it and prints the result. The *newlisp* top-level was inspired by the CMULisp top-level (which was inspired by interlisp) but is much simpler. The top-level is a function (of zero arguments) that can be called by your program. If you prefer another top-level, just redefine the top-level function and type "(reset)" to start running it. The current top-level simply calls the functions tlread, tleval and tlprint to read, evaluate and print. These are supposed to be replaceable by the user. The only one that would make sense to replace is tlprint, which currently uses a function that refuses to go below a certain level and prints "...]" when it finds itself printing a circular list. One might want to prettyprint the results instead. The current top-level numbers the lines that you type to it, and remembers the last n "events" (where n can be set but is defaulted to 25). One can refer to these events in the following "top-level commands":

TOPLEVEL COMMAND SUMMARY

??	prints events - both the input and the result. If you just type "??" you will see all of the recorded events. "?? 3" will show only event 3, and "?? 3 6" will show events 3 through 6.
redo	pretends that you typed the same thing that was typed before. If you type "redo 3" event number 3 is redone. "redo -3" redoes the thing 3 events ago. "redo" is the same as "redo -1".
ed	calls the editor and then does whatever the editor returns. Thus if you want to do event 5 again except for some small change, you can type "ed 5", make the change and leave the editor. "ed -3" and "ed" are analogous to redo.

Finally, you can get the value of event 7 with the function (valueof 7). The other interesting feature of the top-level is that it makes outermost parentheses superfluous for the most part. This works the same way as in CMULisp, so you can use the help for an explanation. If you're not sure and don't want to risk it you can always just include the parentheses.

(top-level)

SIDE EFFECT: *top-level* is the LISP top level function. As well as being the top level function with which the user interacts, it can be called recursively by the user or any function. Thus, the top level can be invoked from inside the editor, break package, or a user function to make its commands available to the user.

NOTE: The CMU FRANZ LISP top-level uses *lineread* rather than *read.* The difference will not usually be noticeable. The principal thing to be careful about is that input to the function or system being called cannot appear on the same line as the top-level call. For example, typing *(editf foo)fP on one line will edit foo and evaluate P, not edit foo and execute the p command in the editor. top-level specially recognizes the following commands:*

(valueof *'g_eventspec*)

RETURNS: the value(s) of the event(s) specified by g_eventspec. If a single event is specified, its value will be returned. If more than one event is specified, or an event has more than one subevent (as for *redo,* etc), a list of vlaues will be returned.

13.2. The File Package

Users typically define functions in lisp and then want to save them for the next session. If you do *(changes)*, a list of the functions that are newly defined or changed will be printed. When you type *(dskouts)*, the functions associated with files will be saved in the new versions of those files. In order to associate functions with files you can either add them to the *filefns* list of an existing file or create a new file to hold them. This is done with the *file* function. If you type *(file new)* the system will create a variable called *newfns.* You may add the names of the functions to go into that file to *newfns.* After you do *(changes)*, the functions which are in no other file are stored in the value of the atom *changes.* To put these all in the new file, *(setq newfns (append newfns changes))*. Now if you do *(changes)*, all of the changed functions should be associated with files. In order to save the changes on the files, do *(dskouts)*. All of the changed files (such as NEW) will be written. To recover the new functions the next time you run FRANZ LISP, do *(dskin new)*.

```
Script started on Sat Mar 14 11:50:32 1981
$ newlisp
Welcome to newlisp...
1.(defun square (x) (* x x))            ; define a new function
square
2.(changes)                                     ; See, this function is associated
                                                ; with no file.
<no-file>     (square)nil
3.(file 'new)                                   ; So let's declare file NEW.
new
4.newfns                                ; It doesn't have anything on it yet.
nil
5.(setq newfns '(square))               ; Add the function associated
(square)                                ; with no file to file NEW.
6.(changes)                                     ; CHANGES magically notices this fact.

new           (square)nil
7.(dskouts)                                     ; We write the file.
creating new
(new)
8.(dskin new)                                   ; We read it in!
(new)
14.Bye
$
script done on Sat Mar 14 11:51:48 1981
```

(changes s_flag)

RETURNS: Changes computes a list containing an entry for each file which defines atoms that have been marked changed. The entry contains the file name and the changed atoms defined therein. There is also a special entry for changes to atoms which are not defined in any known file. The global variable *filelst* contains the list of "known" files. If no flag is passed this result is printed in human readable form and the value returned is t if there were any changes and nil if not. Otherwise nothing is printed and the computer list is returned. The global variable *changes* contains the atoms which are marked changed but not yet associated with any file. The *changes* function attempts to associate these names with files, and any that are not found are considered to belong to no file. The *changes* property is the means by which changed functions are associated with files. When a file is read in or written out its *changes* property is removed.

(dc s_word s_id [g_descriptor1 ...] <text> <esc>**)**

RETURNS: *dc* defines comments. It is exceptional in that its behavior is very context dependent. When *dc* is executed from *dskin* it simply records the fact that the comment exists. It is expected that in interactive mode comments will be found via *getdef* - this allows large comments which do not take up space in your core image. When *dc* is executed from the terminal it expects you to type a comment. *dskout* will write out the comments that you define and also copy the comments on the old version of the file, so that the new version will keep the old comments even though they were never actually brought into core. The optional id is a mechanism for distinguishing among several comments associated with the same word. It defaults to nil. However if you define two comments with the same id, the second is considered to be a replacement for the first. The behavior of *dc* is determined by the value of the global variable *def-comment. def-comment* contains the name of a function that is run. Its arguments are the word, id and attribute list. *def-comment* is initially *dc-define.* Other functions rebind it to *dc-help, dc-userhelp,* and the value of *dskin-comment.* The comment property of an atom is a list of entries, each representing one comment. Atomic entries are assumed to be identifiers of comments on a file but not in core. In-core comments are represented by a list of the id, the attribute list and the comment text. The comment text is an uninterned atom. Comments may be deleted or reordered by editing the comment property.

(dskin l_filenames**)**

SIDE EFFECT: READ-EVAL-PRINTs the contents of the given files. This is the function to use to read files created by *dskout. dskin* also declares the files that it reads (if a *file-fns* list is defined and the file is otherwise declarable by *file*), so that changes to it can be recorded.

(dskout s_file1 ...**)**

SIDE EFFECT: For each file specified, *dskout* assumes the list named filenameFNS (i.e., the file name, excluding extension, concatenated with *fns*) contains a list of function names, etc., to be loaded Any previous version of the file will be renamed to have extension ".back".

(dskouts s_file1 ...**)**

SIDE EFFECT: applies *dskout* to and prints the name of each s_filei (with no additional arguments, assuming filenameFNS to be a list to be loaded) for which s_file*i* is either not in *filelst* (meaning it is a new file not previously declared by *file* or given as an argument to *dskin, dskouts,* or *dskouts*) or is in *filelst* and has some recorded changes to definitions of atoms in filenameFNS, as recorded by *mark!changed* and noted by changes. If *filei* is not specified, *filelst* will be used. This is the most common way of using dskouts. Typing *(dskouts)* will save every file reported by *(changes)* to have changed definitions.

(dv s_atom g_value**)**

EQUIVALENT TO:*(setq atom 'value). dv* calls *mark!changed.*

(file 's_file**)**

SIDE EFFECT: declares its argument to be a file to be used for reporting and saving changes to functions by adding the file name to a list of files, *filelst. file* is called for each file argument of *dskin, dskout,* and *dskouts.*

(file-fns 's_file**)**

RETURNS:the name of the fileFNS list for its file argument s_file.

(getdef 's_file ['s_i1 ...]**)**

SIDE EFFECT: selectively executes definitions for atoms s_i1 ... from the specified file. Any of the words to be defined which end with "@" will be treated as patterns in which the @ matchs any suffix (just like the editor). *getdef* is driven by *getdeftable* (and thus may be programmed). It looks for lines in the file that start with a word in the table. The first character must be a "(" or "[" followed by the word, followed by a space, return or something else that will not be considered as part of the identifier by *read,* e.g., "(" is unacceptable. When one is found the next word is read. If it matches one of the identifiers in the call to *getdef* then the table entry is executed. The table entry is a function of the expression starting in this line. Output from *dskout* is in acceptable format for *getdef. getdef*

RETURNS:a list of the words which match the ones it looked for, for which it found (but, depending on the table, perhaps did not execute) in the file.

NOTE: *getdeftable* is the table that drives *getdef.* It is in the form of an association list. Each element is a dotted pair consisting of the name of a function for which *getdef* searches and a function of one argument to be executed when it is found.

(mark!changed 's_f**)**

SIDE EFFECT: records the fact that the definition of s_f has been changed. It is automatically called by *def, defun, de, df, defprop, dm, dv,* and the editor when a definition is altered.

CHAPTER 14

The LISP Stepper

14.1. Simple Use Of Stepping

(step s_arg1...)

NOTE: The LISP "stepping" package is intended to give the LISP programmer a facility analogous to the Instruction Step mode of running a machine language program. The user interface is through the function (fexpr) step, which sets switches to put the LISP interpreter in and out of "stepping" mode. The most common *step* invocations follow. These invocations are usually typed at the top-level, and will take effect immediately (i.e. the next S-expression typed in will be evaluated in stepping mode).

```
(step t)                    ; Turn on stepping mode.
(step nil)                  ; Turn off stepping mode.
```

SIDE EFFECT: In stepping mode, the LISP evaluator will print out each S-exp to be evaluated before evaluation, and the returned value after evaluation, calling itself recursively to display the stepped evaluation of each argument, if the S-exp is a function call. In stepping mode, the evaluator will wait after displaying each S-exp before evaluation for a command character from the console.

STEP COMMAND SUMMARY

<return>	Continue stepping recursively.
c	Show returned value from this level only, and continue stepping upward.
e	Only step interpreted code.
g	Turn off stepping mode. (but continue evaluation without stepping).
n <number>	Step through <number> evaluations without stopping
p	Redisplay current form in full (i.e. rebind prinlevel and prinlength to nil)
b	Get breakpoint
q	Quit
d	Call debug

14.2. Advanced Features

14.2.1. Selectively Turning On Stepping.

If

(step foo1 foo2 ...)

is typed at top level, stepping will not commence immediately, but rather when the evaluator first encounters an S-expression whose car is one of *foo1, foo2*, etc. This form will then display at the console, and the evaluator will be in stepping mode waiting for a command character.

Normally the stepper intercepts calls to *funcall* and *eval*. When *funcall* is intercepted, the arguments to the function have already been evaluated but when *eval* is intercepted, the arguments have not been evaluated. To differentiate the two cases, when printing the form in evaluation, the stepper preceded intercepted calls to *funcall* with "f:". Calls to *funcall* are normally caused by compiled lisp code calling other functions, whereas calls to *eval* usually occur when lisp code is interpreted. To step only calls to eval use: *(step e)*

14.2.2. Stepping With Breakpoints.

For the moment, step is turned off inside of error breaks, but not by the break function. Upon exiting the error, step is reenabled. However, executing *(step nil)* inside a error loop will turn off stepping globally, i.e. within the error loop, and after return has be made from the loop.

14.3. Overhead of Stepping.

If stepping mode has been turned off by *(step nil)*, the execution overhead of having the stepping packing in your LISP is identically nil. If one stops stepping by typing "g", every call to eval incurs a small overhead--several machine instructions, corresponding to the compiled code for a simple cond and one function pushdown. Running with *(step foo1 foo2 ...)* can be more expensive, since a member of the car of the current form into the list *(foo1 foo2 ...)* is required at each call to eval.

14.4. Evalhook and Funcallhook

There are hooks in the FRANZ LISP interpreter to permit a user written function to gain control of the evaluation process. These hooks are used by the Step package just described. There are two hooks and they have been strategically placed in the two key functions in the interpreter: *eval* (which all interpreted code goes through) and *funcall* (which all compiled code goes through if *(sstatus translink nil)* has been done). The hook in *eval* is compatible with Maclisp, but there is no Maclisp equivalent of the hook in *funcall*.

To arm the hooks two forms must be evaluated: *(*rset t)* and *(sstatus evalhook t)*. Once that is done, *eval* and *funcall* do a special check when they enter.

If *eval* is given a form to evaluate, say *(foo bar)*, and the symbol 'evalhook' is non nil, say its value is 'ehook', then *eval* will lambda bind the symbols 'evalhook' and 'funcallhook' to nil and will call ehook passing *(foo bar)* as the argument. It is ehook's responsibility to evaluate *(foo bar)* and return its value. Typically ehook will call the function 'evalhook' to evaluate *(foo bar)*. Note that 'evalhook' is a symbol whose function binding is a system function described in Chapter 4, and whose value binding, if non nil, is the name of a user written function (or a lambda expression, or a binary object) which will gain control whenever eval is called. 'evalhook' is also the name of the *status* tag which must be set for all of this to work.

If *funcall* is given a function, say foo, and a set of already evaluated arguments, say barv and bazv, and if the symbol 'funcallhook' has a non nil value, say 'fhook', then *funcall* will lambda bind 'evalhook' and 'funcallhook' to nil and will call fhook with arguments barv, bazv and foo. Thus fhook must be a lexpr since it may be given any number of arguments. The function to call, foo in this case, will be the *last* of the arguments given to fhook. It is fhooks responsibility to do the function call and return the value. Typically fhook will call the function *funcallhook* to do the funcall. This is an example of a funcallhook function which just prints the arguments on each entry to funcall and the return value.

```
-> (defun fhook n (let ((form (cons (arg n) (listify (1- n))))
                                (retval))
                           (patom "calling ")(print form)(terpr)
                           (setq retval (funcallhook form 'fhook))
                           (patom "returns ")(print retval)(terpr)
                           retval))
fhook
-> (*rset t) (sstatus evalhook t) (sstatus translink nil)
-> (setq funcallhook 'fhook)
calling (print fhook)                    ;; now all compiled code is traced
fhookreturns nil
calling (terpr)

returns nil
calling (patom "-> ")
-> returns "-> "
calling (read nil Q00000)
(array foo t 10)                ;; to test it, we see what happens when
returns (array foo t 10)        ;; we make an array
calling (eval (array foo t 10))
calling (append (10) nil)
returns (10)
calling (lessp 1 1)
returns nil
calling (apply times (10))
returns 10
calling (small-segment value 10)
calling (boole 4 137 127)
returns 128
 ... there is plenty more ...
```

CHAPTER 15

The FIXIT Debugger

15.1. Introduction FIXIT is a debugging environment for FRANZ LISP users doing program development. This documentation and FIXIT were written by David S. Touretzky of Carnegie-Mellon University for MACLisp, and adapted to FRANZ LISP by Mitch Marcus of Bell Labs. One of FIXIT's goals is to get the program running again as quickly as possible. The user is assisted in making changes to his functions "on the fly", i.e. in the midst of execution, and then computation is resumed.

To enter the debugger type *(debug)*. The debugger goes into its own read-eval-print loop. Like the top-level, the debugger understands certain special commands. One of these is help, which prints a list of the available commands. The basic idea is that you are somewhere in a stack of calls to eval. The command "bka" is probably the most appropriate for looking at the stack. There are commands to move up and down. If you want to know the value of "x" as of some place in the stack, move to that place and type "x" (or (cdr x) or anything else that you might want to evaluate). All evaluation is done as of the current stack position. You can fix the problem by changing the values of variables, editing functions or expressions in the stack etc. Then you can continue from the current stack position (or anywhere else) with the "redo" command. Or you can simply return the right answer with the "return" command.

When it is not immediately obvious why an error has occurred or how the program got itself into its current state, FIXIT comes to the rescue by providing a powerful debugging loop in which the user can:

- examine the stack
- evaluate expressions in context
- enter stepping mode
- restart the computation at any point

The result is that program errors can be located and fixed extremely rapidly, and with a minimum of frustration.

The debugger can only work effectively when extra information is kept about forms in evaluation by the lisp system. Evaluating *(*rset t)* tells the lisp system to maintain this information. If you are debugging compiled code you should also be sure that the compiled code to compiled code linkage tables are unlinked, i.e do *(sstatus translink nil)*.

(debug [s_msg])

NOTE: Within a program, you may enter a debug loop directly by putting in a call to *debug* where you would normally put a call to *break*. Also, within a break loop you may enter FIXIT by typing *debug*. If an argument is given to DEBUG, it is treated as a message to be printed before the debug loop is entered. Thus you can put *(debug |just before loop|)* into a program to indicate what part of the program is being debugged.

FIXIT Command Summary

TOP go to top of stack (latest expression)
BOT go to bottom of stack (first expression)
P show current expression (with ellipsis)
PP show current expression in full
WHERE give current stack position
HELP types the abbreviated command summary found
in /usr/lisp/doc/fixit.help. H and ? work too.
U go up one stack frame
U n go up n stack frames
U f go up to the next occurrence of function f
U n f go up n occurrences of function f
UP go up to the next user-written function
UP n go up n user-written functions
...the DN and DNFN commands are similar, but go down
...instead of up.
OK resume processing; continue after an error or debug loop
REDO restart the computation with the current stack frame.
The OK command is equivalent to TOP followed by REDO.
REDO f restart the computation with the last call to function f.
(The stack is searched downward from the current position.)
STEP restart the computation at the current stack frame,
but first turn on stepping mode. (Assumes Rich stepper is loaded.)
RETURN e return from the current position in the computation
with the value of expression e.
BK.. print a backtrace. There are many backtrace commands,
formed by adding suffixes to the BK command. "BK" gives
a backtrace showing only user-written functions, and uses
ellipsis. The BK command may be suffixed by one or more
of the following modifiers:
..F.. show function names instead of expressions
..A.. show all functions/expressions, not just user-written ones
..V.. show variable bindings as well as functions/expressions
..E.. show everything in the expression, i.e. don't use ellipsis
..C.. go no further than the current position on the stack
Some of the more useful combinations are BKFV, BKFA,
and BKFAV.
BK.. n show only n levels of the stack (starting at the top).
(BK n counts only user functions; BKA n counts all functions.)
BK.. f show stack down to first call of function f
BK.. n f show stack down to nth call of function f

15.2. Interaction with *trace* FIXIT knows about the standard Franz trace package, and tries to make tracing invisible while in the debug loop. However, because of the way *trace* works, it may sometimes be the case that the functions on the stack are really un*interned* atoms that have the same name as a traced function. (This only happens when a function is traced WHEREIN another one.) FIXIT will call attention to *trace's* hackery by printing an appropriate tag next to these stack entries.

15.3. Interaction with *step* The *step* function may be invoked from within FIXIT via the STEP command. FIXIT initially turns off stepping when the debug loop is entered. If you step through a function and get an error, FIXIT will still be invoked normally. At any time during stepping, you may explicitly enter FIXIT via the "D" (debug) command.

15.4. Multiple error levels FIXIT will evaluate arbitrary LISP expressions in its debug loop. The evaluation is not done within an *errset*, so, if an error occurs, another invocation of the debugger can be made. When there are multiple errors on the stack, FIXIT displays a barrier symbol between each level that looks something like <-------UDF-->. The UDF in this case stands for UnDefined Function. Thus, the upper level debug loop was invoked by an undefined function error that occurred while in the lower loop.

CHAPTER 16

The LISP Editor

16.1. The Editors

It is quite possible to use VI, Emacs or other standard editors to edit your lisp programs, and many people do just that. However there is a lisp structure editor which is particularly good for the editing of lisp programs, and operates in a rather different fashion, namely within a lisp environment. application. It is handy to know how to use it for fixing problems without exiting from the lisp system (e.g. from the debugger so you can continue to execute rather than having to start over.) The editor is not quite like the top-level and debugger, in that it expects you to type editor commands to it. It will not evaluate whatever you happen to type. (There is an editor command to evaluate things, though.)

The editor is available (assuming your system is set up correctly with a lisp library) by typing (load 'cmufncs) and (load 'cmuedit).

The most frequent use of the editor is to change function definitions by starting the editor with one of the commands described in section 16.14. (see *editf*), values (*editv*), properties (*editp*), and expressions (*edite*). The beginner is advised to start with the following (very basic) commands: *ok, undo, p,* #, under which are explained two different basic commands which start with numbers, and f.

This documentation, and the editor, were imported from PDP-10 CMULisp by Don Cohen. PDP-10 CMULisp is based on UCILisp, and the editor itself was derived from an early version of Interlisp. Lars Ericson, the author of this section, has provided this very concise summary. Tutorial examples and implementation details may be found in the Interlisp Reference Manual, where a similar editor is described.

16.2. Scope of Attention

Attention-changing commands allow you to look at a different part of a Lisp expression you are editing. The sub-structure upon which the editor's attention is centered is called "the current expression". Changing the current expression means shifting attention and not actually modifying any structure.

SCOPE OF ATTENTION COMMAND SUMMARY

n (n>0) . Makes the nth element of the current expression be the new current expression.

-n (n>0). Makes the nth element from the end of the current expression be the new current expression.

0. Makes the next higher expression be the new correct expression. If the intention is to go back to the next higher left parenthesis, use the command !0.

up . If a p command would cause the editor to type ... before typing the current expression, (the current expression is a tail of the next higher expression) then has no effect; else, up makes the old current expression the first element in the new current expression.

!0 . Goes back to the next higher left parenthesis.

^ . Makes the top level expression be the current expression.

nx . Makes the current expression be the next expression.

(nx n) equivalent to n nx commands.

!nx . Makes current expression be the next expression at a higher level. Goes through any number of right parentheses to get to the next expression.

bk . Makes the current expression be the previous expression in the next higher expression.

(nth n) n>0 . Makes the list starting with the nth element of the current expression be the current expression.

(nth $) - generalized nth command. nth locates $, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation.

:: . (pattern :: . $) e.g., (cond :: return). finds a cond that contains a return, at any depth.

(below com x) . The below command is useful for locating a substructure by specifying something it contains. (below cond) will cause the cond clause containing the current expression to become the new current expression. Suppose you are editing a list of lists, and want to find a sublist that contains a foo (at any depth). Then simply executes f foo (below).

(nex x) . same as *(below x)* followed by nx. For example, if you are deep inside of a selectq clause, you can advance to the next clause with *(nex selectq).*

nex. The atomic form of *nex* is useful if you will be performing repeated executions of *(nex x).* By simply marking the chain corresponding to x, you can use *nex* to step through the sublists.

16.3. Pattern Matching Commands

Many editor commands that search take patterns. A pattern *pat* matches with x if:

PATTERN SPECIFICATION SUMMARY

- *pat* is *eq* to x.

- *pat* is &.

- *pat* is a number and equal to x.

- if (car *pat*) is the atom *any*, (cdr *pat*) is a list of patterns, and *pat* matches x if and only if one of the patterns on (cdr *pat*) matches x.

- if *pat* is a literal atom or string, and (nthchar *pat* -1) is @, then *pat* matches with any literal atom or string which has the same initial characters as *pat*, e.g. ver@ matches with verylongatom, as well as "verylongstring".

- if (car *pat*) is the atom –, *pat* matches x if (a) (cdr *pat*)=nil, i.e. *pat*=(–), e.g., (a –) matches (a) (a b c) and (a . b) in other words, – can match any tail of a list. (b) (cdr *pat*) matches with some tail of x, e.g. (a – (&)) will match with (a b c (d)), but not (a b c d), or (a b c (d) e). however, note that (a – (&) –) will match with (a b c (d) e). in other words, – will match any interior segment of a list.

- if (car *pat*) is the atom ==, *pat* matches x if and only if (cdr *pat*) is *eq* to x. (this pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command typed in by the user obviously cannot be *eq* to existing structure.) - otherwise if x is a list, *pat* matches x if (car *pat*) matches (car x), and (cdr *pat*) matches (cdr x).

- when searching, the pattern matching routine is called only to match with elements in the structure, unless the pattern begins with :::, in which case cdr of the pattern is matched against tails in the structure. (in this case, the tail does not have to be a proper tail, e.g. (::: a –) will match with the element (a b c) as well as with cdr of (x a b c), since (a b c) is a tail of (a b c).)

16.3.1. Commands That Search

SEARCH COMMAND SUMMARY

f pattern . f informs the editor that the next command is to be interpreted as a pattern. If no pattern is given on the same line as the f then the last pattern is used. f pattern means find the next instance of pattern.

(f pattern n). Finds the next instance of pattern.

(f pattern t). similar to f pattern, except, for example, if the current expression is (cond ..), f cond will look for the next cond, but (f cond t) will 'stay here'.

(f pattern n) n>0. Finds the nth place that pattern matches. If the current expression is (foo1 foo2 foo3), (f f00@ 3) will find foo3.

(f pattern) or (f pattern nil). only matches with elements at the top level of the current expression. If the current expression is *(prog nil (setq x (cond & &)) (cond &) ...)* f (cond –) will find the cond inside the setq, whereas (f (cond –)) will find the top level cond, i.e., the second one.

(second . $) . same as (lc . $) followed by another (lc . $) except that if the first succeeds and second fails, no change is made to the edit chain.

(third . $) . Similar to second.

(fs pattern1 ... patternn) . equivalent to f pattern1 followed by f pattern2 ... followed by f pattern n, so that if f

pattern m fails, edit chain is left at place pattern m-1 matched.

(f= expression x) . Searches for a structure eq to expression.

(orf pattern1 ... patternn) . Searches for an expression that is matched by either pattern1 or ... patternn.

bf pattern . backwards find. If the current expression is *(prog nil (setq x (setq y (list z))) (cond ((setq w --) --)) --)* f list followed by bf setq will leave the current expression as (setq y (list z)), as will f cond followed by bf setq

(bf pattern t). backwards find. Search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc.

16.3.1.1. Location Specifications Many editor commands use a method of specifying position called a location specification. The meta-symbol $ is used to denote a location specification. $ is a list of commands interpreted as described above. $ can also be atomic, in which case it is interpreted as (list $). a location specification is a list of edit commands that are executed in the normal fashion with two exceptions. first, all commands not recognized by the editor are interpreted as though they had been preceded by f. The location specification (cond 2 3) specifies the 3rd element in the first clause of the next cond.

the if command and the ## function provide a way of using in location specifications arbitrary predicates applied to elements in the current expression.

In insert, delete, replace and change, if $ is nil (empty), the corresponding operation is performed on the current edit chain, i.e. (replace with (car x)) is equivalent to (:(car x)). for added readability, here is also permitted, e.g., (insert (print x) before here) will insert (print x) before the current expression (but not change the edit chain). It is perfectly legal to ascend to insert, replace, or delete. for example (insert (*return*) after ^ prog -1) will go to the top, find the first prog, and insert a (*return*) at its end, and not change the current edit chain.

The a, b, and : commands all make special checks in e1 thru em for expressions of the form (## . coms). In this case, the expression used for inserting or replacing is a copy of the current expression after executing coms, a list of edit commands. (insert (## f cond -1 -1) after3) will make a copy of the last form in the last clause of the next cond, and insert it after the third element of the current expression.

$. In descriptions of the editor, the meta-symbol $ is used to denote a location specification. $ is a list of commands interpreted as described above. $ can also be atomic.

LOCATION COMMAND SUMMARY

(lc . $) . Provides a way of explicitly invoking the location operation. (lc cond 2 3) will perform search.

(lcl . $) . Same as lc except search is confined to current expression. To find a cond containing a *return*, one might use the location specification (cond (lcl *return*)) where the would reverse the effects of the lcl command, and make

the final current expression be the cond.

16.3.2. The Edit Chain The edit-chain is a list of which the first element is the the one you are now editing ("current expression"), the next element is what would become the current expression if you were to do a 0, etc., until the last element which is the expression that was passed to the editor.

EDIT CHAIN COMMAND SUMMARY

mark . Adds the current edit chain to the front of the list marklst.

_ . Makes the new edit chain be (car marklst).

(_ pattern) . Ascends the edit chain looking for a link which matches pattern. for example:

__ . Similar to _ but also erases the mark.

\ . Makes the edit chain be the value of unfind. unfind is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ^, _, __, !nx, all commands that involve a search, e.g., f, lc, ::, below, et al and and themselves. if the user types f cond, and then f car, would take him back to the cond. another would take him back to the car, etc.

\p . Restores the edit chain to its state as of the last print operation. If the edit chain has not changed since the last printing, \p restores it to its state as of the printing before that one. If the user types p followed by 3 2 1 p, \p will return to the first p, i.e., would be equivalent to 0 0 0. Another \p would then take him back to the second p.

16.4. Printing Commands

PRINTING COMMAND SUMMARY

p Prints current expression in abbreviated form. (p m) prints mth element of current expression in abbreviated form. (p m n) prints mth element of current expression as though printlev were given a depth of n. (p 0 n) prints current expression as though printlev were given a depth of n. (p cond 3) will work.

? . prints the current expression as though printlev were given a depth of 100.

pp . pretty-prints the current expression.

*pp** . is like pp, but forces comments to be shown.

16.5. Structure Modification Commands

All structure modification commands are undoable. See *undo*.

STRUCTURE MODIFICATION COMMAND SUMMARY

 # [editor commands] (n) n>1 deletes the corresponding element from the current expression.

(n e1 ... em) n,m>1 replaces the nth element in the current expression with e1 ... em.

(-n e1 ... em) n,m>1 inserts e1 ... em before the n element in the current expression.

(n e1 ... em) (the letter "n" for "next" or "nconc", not a number) m>1 attaches e1 ... em at the end of the current expression.

(a e1 ... em) . inserts e1 ... em after the current expression (or after its first element if it is a tail).

(b e1 ... em) . inserts e1 ... em before the current expression. to insert foo before the last element in the current expression, perform -1 and then (b foo).

(: e1 ... em) . replaces the current expression by e1 ... em. If the current expression is a tail then replace its first element.

delete or (:) . deletes the current expression, or if the current expression is a tail, deletes its first element.

(delete . $). does a (lc . $) followed by delete. current edit chain is not changed.

(insert e1 ... em before . $) . similar to (lc. $) followed by (b e1 ... em).

(insert e1 ... em after . $). similar to insert before except uses a instead of b.

(insert e1 ... em for . $). similar to insert before except uses : for b.

(replace $ with e1 ... em) . here $ is the segment of the command between replace and with.

(change $ to e1 ... em) . same as replace with.

16.6. Extraction and Embedding Commands

EXTRACTION AND EMBEDDING COMMAND SUMMARY

(xtr . $) . replaces the original current expression with the expression that is current after performing (lcl . $).

(mbd x) . x is a list, substitutes the current expression for all instances of the atom * in x, and replaces the current expression with the result of that substitution. (mbd x) : x atomic, same as (mbd (x *)).

(extract $1 from $2) . extract is an editor command which replaces the current expression with one of its subexpressions (from any depth). ($1 is the segment between extract and from.) example: if the current expression is (print (cond ((null x) y) (t z))) then following (extract y from cond), the current expression will be (print y). (extract 2 -1 from cond), (extract y from 2), (extract 2 -1 from 2) will all produce the same result.

(embed $ in . x) . embed replaces the current expression with a new expression which contains it as a subexpression.

($ is the segment between embed and in.) example: (embed print in setq x), (embed 3 2 in *return*), (embed cond 3 1 in (or * (null x))).

16.7. Move and Copy Commands

MOVE AND COPY COMMAND SUMMARY

(move $1 to com . $2) . ($1 is the segment between move and to.) where com is before, after, or the name of a list command, e.g., :, n, etc. If $2 is nil, or (here), the current position specifies where the operation is to take place. If $1 is nil, the move command allows the user to specify some place the current expression is to be moved to. if the current expression is (a b d c), (move 2 to after 4) will make the new current expression be (a c d b).

(mv com . $) . is the same as (move here to com . $).

(copy $1 to com . $2) is like move except that the source expression is not deleted.

(cp com . $). is like mv except that the source expression is not deleted.

16.8. Parentheses Moving Commands The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses.

PARENTHESES MOVING COMMAND SUMMARY

(bi n m) . both in. inserts parentheses before the nth element and after the mth element in the current expression. example: if the current expression is (a b (c d e) f g), then (bi 2 4) will modify it to be (a (b (c d e) f) g). (bi n) : same as (bi n n). example: if the current expression is (a b (c d e) f g), then (bi -2) will modify it to be (a b (c d e) (f) g).

(bo n) . both out. removes both parentheses from the nth element. example: if the current expression is (a b (c d e) f g), then (bo d) will modify it to be (a b c d e f g).

(li n) . left in. inserts a left parenthesis before the nth element (and a matching right parenthesis at the end of the current expression). example: if the current expression is (a b (c d e) f g), then (li 2) will modify it to be (a (b (c d e) f g)).

(lo n) . left out. removes a left parenthesis from the nth element. all elements following the nth element are deleted. example: if the current expression is (a b (c d e) f g), then (lo 3) will modify it to be (a b c d e).

(ri n m) . right in. move the right parenthesis at the end of the nth element in to after the mth element. inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression. example: if the current expression is (a (b c d e) f g), (ri 2 2) will modify it to be (a (b c) d e f g).

(ro n) . right out. move the right parenthesis at the end of the nth element out to the end of the current expression.

removes the right parenthesis from the nth element, moving it to the end of the current expression. all elements following the nth element are moved inside of the nth element. example: if the current expression is (a b (c d e) f g), (ro 3) will modify it to be (a b (c d e f g)).

(r x y) replaces all instances of x by y in the current expression, e.g., (r caadr cadar). x can be the s-expression (or atom) to be substituted for, or can be a pattern which specifies that s-expression (or atom).

(sw n m) switches the nth and mth elements of the current expression. for example, if the current expression is (list (cons (car x) (car y)) (cons (cdr y))), (sw 2 3) will modify it to be (list (cons (cdr x) (cdr y)) (cons (car x) (car y))). (sw car cdr) would produce the same result.

16.8.1. Using to and thru

to, thru, extract, embed, delete, replace, and move can be made to operate on several contiguous elements, i.e., a segment of a list, by using the to or thru command in their respective location specifications. thru and to are intended to be used in conjunction with extract, embed, delete, replace, and move. to and thru can also be used directly with xtr (which takes after a location specification), as in (xtr (2 thru 4)) (from the current expression).

TO AND THRU COMMAND SUMMARY

($1 to $2) . same as thru except last element not included.

($1 to). same as ($1 thru -1)

($1 thru $2) . If the current expression is (a (b (c d) (e) (f g h) i) j k), following (c thru g), the current expression will be ((c d) (e) (f g h)). If both $1 and $2 are numbers, and $2 is greater than $1, then $2 counts from the beginning of the current expression, the same as $1. in other words, if the current expression is (a b c d e f g), (3 thru 4) means (c thru d), not (c thru f). in this case, the corresponding bi command is (bi 1 $2-$1+1).

($1 thru). same as *($1 thru -1).*

16.9. Undoing Commands each command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command. The undo command undoes the last, i.e., most recent such command.

UNDO COMMAND SUMMARY

undo . the undo command undoes most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., mbd undone. The edit chain is then exactly what it was before the 'undone' command had been performed.

!undo . undoes all modifications performed during this editing session, i.e., this call to the editor.

unblock . removes an undo-block. If executed at a non-blocked state, i.e., if undo or !undo could operate, types not blocked.

test . adds an undo-block at the front of undolst. note that test together with !undo provide a 'tentative' mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !undo command.

undolst [value]. each editor command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command.

?? prints the entries on undolst. The entries are listed most recent entry first.

16.10. Commands that Evaluate

EVALUATION COMMAND SUMMARY

e . only when typed in, (i.e., (insert d before e) will treat e as a pattern) causes the editor to call the lisp interpreter giving it the next input as argument.

(e x) evaluates x, and prints the result. (e x t) same as (e x) but does not print.

(i c x1 ... xn) same as (c y1 ... yn) where yi=(eval xi). example: (i 3 (cdr foo)) will replace the 3rd element of the current expression with the cdr of the value of foo. (i n foo (car fie)) will attach the value of foo and car of the value of fie to the end of the current expression. (i f= foo t) will search for an expression eq to the value of foo. If c is not an atom, it is evaluated as well.

(coms x1 ... xn) . each xi is evaluated and its value executed as a command. The i command is not very convenient for computing an entire edit command for execution, since it computes the command name and its arguments separately. also, the i command cannot be used to compute an atomic command. The coms and comsq commands provide more general ways of computing commands. (coms (cond (x (list 1 x)))) will replace the first element of the current expression with the value of x if non-nil, otherwise do nothing. (nil as a command is a nop.)

(comsq com1 ... comn) . executes com1 ... comn. comsq is mainly useful in conjunction with the coms command. for example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the coms command. he would then write (coms (cons (quote comsq) x)) where x computed the list of commands, e.g., (coms (cons (quote comsq) (get foo (quote commands))))

16.11. Commands that Test

TESTING COMMAND SUMMARY

(if x) generates an error unless the value of (eval x) is non-nil, i.e., if (eval x) causes an error or (eval x)=nil, if will cause an error. (if x coms1 coms2) if (eval x) is non-nil, execute coms1; if (eval x) causes an error or is equal to nil, execute coms2. (if x coms1) if (eval x) is non-nil, execute coms1; otherwise generate an error.

(lp . coms) . repeatedly executes coms, a list of commands, until an error occurs. (lp f print (n t)) will attach

a t at the end of every print expression. (lp f print (if (## 3) nil ((n t)))) will attach a t at the end of each print expression which does not already have a second argument. (i.e. the form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((n t)) as the list of commands to be executed. The if could also be written as (if (cddr (##)) nil ((n t))).).

(lpq . coms) same as lp but does not print n occurrences.

(orr coms1 ... comsn) . orr begins by executing coms1, a list of commands. If no error occurs, orr is finished. otherwise, orr restores the edit chain to its original value, and continues by executing coms2, etc. If none of the command lists execute without errors, i.e., the orr "drops off the end", orr generates an error. otherwise, the edit chain is left as of the completion of the first command list which executes without error.

16.12. Editor Macros

Many of the more sophisticated branching commands in the editor, such as orr, if, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire. (however, built in commands always take precedence over macros, i.e., the editor's repertoire can be expanded, but not modified.) macros are defined by using the m command.

(m c . coms) for c an atom, m defines c as an atomic command. (if a macro is redefined, its new definition replaces its old.) executing c is then the same as executing the list of commands coms. macros can also define list commands, i.e., commands that take arguments. (m (c) (arg[1] ... arg[n]) . coms) c an atom. m defines c as a list command. executing (c e1 ... en) is then performed by substituting e1 for arg[1], ... en for arg[n] throughout coms, and then executing coms. a list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments'. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. if the of arguments. (m (c) args . coms) c, args both atoms, defines c as a list command. executing (c e1 ... en) is performed by substituting (e1 ... en), i.e., cdr of the command, for args throughout coms, and then executing coms.

(m bp bk up p) will define bp as an atomic command which does three things, a bk, an up, and a p. note that macros can use commands defined by macros as well as built in commands in their definitions. for example, suppose z is defined by (m z -1 (if (null (##)) nil (p))), i.e. z does a -1, and then if the current expression is not nil, a p. now we can define zz by (m zz -1 z), and zzz by (m zzz -1 -1 z) or (m zzz -1 zz). we could define a more general bp by (m (bp) (n) (bk n) up p). (bp 3) would perform (bk 3), followed by an up, followed by a p. The command second can be defined as a macro by (m (2nd) x (orr ((lc . x) (lc . x)))).

Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are completely independent. in other words, the existence of an atomic definition for c in no way affects the treatment of c when it appears as car of a list command, and the existence of a list definition for c in no way affects the treatment of c when it appears as an atom. in particular, c can be used as the name of either an atomic command, or a list command, or both. in the latter case, two entirely different definitions can be used. note also

that once c is defined as an atomic command via a macro definition, it will not be searched for when used in a location specification, unless c is preceded by an f. (insert -- before bp) would not search for bp, but instead perform a bk, an up, and a p, and then do the insertion. The corresponding also holds true for list commands.

(bind . coms) bind is an edit command which is useful mainly in macros. it binds three dummy variables #1, #2, #3, (initialized to nil), and then executes the edit commands coms. note that these bindings are only in effect while the commands are being executed, and that bind can be used recursively; it will rebind #1, #2, and #3 each time it is invoked.

usermacros [value]. this variable contains the users editing macros . if you want to save your macros then you should save usermacros. you should probably also save editcomsl.

editcomsl [value]. editcomsl is the list of "list commands" recognized by the editor. (these are the ones of the form (command arg1 arg2 ...).)

16.13. Miscellaneous Editor Commands

MISCELLANEOUS EDITOR COMMAND SUMMARY

ok . Exits from the editor.

nil . Unless preceded by f or bf, is always a null operation.

tty: . Calls the editor recursively. The user can then type in commands, and have them executed. The tty: command is completed when the user exits from the lower editor (with ok or stop). the tty: command is extremely useful. it enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. for example the command (move 3 to after cond 3 p tty:) allows the user to interact, in effect, within the move command. he can verify for himself that the correct location has been found, or complete the specification "by hand". in effect, tty: says "I'll tell you what you should do when you get there."

stop . exits from the editor with an error. mainly for use in conjunction with tty: commands that the user wants to abort. since all of the commands in the editor are errset protected, the user must exit from the editor via a command. stop provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session.

tl . tl calls (top-level). to return to the editor just use the *return* top-level command.

repack . permits the 'editing' of an atom or string.

(repack $) does (lc . $) followed by repack, e.g. (repack this@).

(makefn form args n m) . makes (car form) an expr with the nth through mth elements of the current expression with each occurrence of an element of (cdr form) replaced by the corresponding element of args. The nth through mth elements are replaced by form.

(makefn form args n). same as (makefn form args n n).

(s var . $) . sets var (using setq) to the current expression after performing (lc . $). (s foo) will set foo to the current expression, (s foo -1 1) will set foo to the first element in the last element of the current expression.

16.14. Editor Functions

(editf s_x1 ...**)**

SIDE EFFECT: edits a function. s_x1 is the name of the function, any additional arguments are an optional list of commands.

RETURNS:s_x1.

NOTE: if s_x1 is not an editable function, editf generates an fn not editable error.

(edite l_expr l_coms s_atm**))**
edits an expression. its value is the last element of (editl (list l_expr) l_coms s_atm nil nil).

(editracefn s_com**)**
is available to help the user debug complex edit macros, or subroutine calls to the editor. editracefn is to be defined by the user. whenever the value of editracefn is non-nil, the editor calls the function editracefn before executing each command (at any level), giving it that command as its argument. editracefn is initially equal to nil, and undefined.

(editv s_var [g_com1 ...]**)**

SIDE EFFECT: similar to editf, for editing values. editv sets the variable to the value returned.

RETURNS:the name of the variable whose value was edited.

(editp s_x**)**

SIDE EFFECT: similar to editf for editing property lists. used if x is nil.

RETURNS:the atom whose property list was edited.

(editl coms atm marklst mess**)**

SIDE EFFECT: editl is the editor. its first argument is the edit chain, and its value is an edit chain, namely the value of l at the time editl is exited. (l is a special variable, and so can be examined or set by edit commands. ^ is equivalent to (e (setq l(last l)) t).) coms is an optional list of commands. for interactive editing, coms is nil. in this case, editl types edit and then waits for input from the teletype. (if mess is not nil editl types it instead of edit. for example, the tty: command is essentially (setq l (editl l nil nil (quote tty:))).) exit occurs only via an ok, stop, or save command. If coms is not nil, no message is typed, and each member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed , the rest of the commands are ignored, and editl exits with an error, i.e., the effect is the same as though a stop command had been executed. If all commands execute successfully, editl returns the current value of l. marklst is the list of marks. on calls from editf, atm is the name of the function being edited; on calls from editv, the name of the variable, and calls from editp, the atom of which some property of its property list is being edited. The property list

of atm is used by the save command for saving the state of the edit. save will not save anything if atm=nil i.e., when editing arbitrary expressions via edite or editl directly.

(editfns s_x [g_coms1 ...])
fsubr function, used to perform the same editing operations on several functions. editfns maps down the list of functions, prints the name of each function, and calls the editor (via editf) on that function.

EXAMPLE:editfns foofns (r fie fum)) will change every fie to fum in each of the functions on foofns.

NOTE: the call to the editor is errset protected, so that if the editing of one function causes an error, editfns will proceed to the next function. in the above example, if one of the functions did not contain a fie, the r command would cause an error, but editing would continue with the next function. The value of editfns is nil.

(edit4e pat y)

SIDE EFFECT: is the pattern match routine.

RETURNS:t if pat matches y. see edit-match for definition of 'match'.

NOTE: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings that end in at-signs. These are replaced by patterns of the form (cons (quote /@) (explodec atom)). from the standpoint of edit4e, pattern type 5, atoms or strings ending in at-signs, is really "if car[pat] is the atom @ (at-sign), pat will match with any literal atom or string whose initial character codes (up to the @) are the same as those in cdr[pat]." if the user wishes to call edit4e directly, he must therefore convert any patterns which contain atoms or strings ending in at-signs to the form recognized by edit4e. this can be done via the function editfpat.

(editfpat pat flg)
makes a copy of pat with all patterns of type 5 (see edit-match) converted to the form expected by edit4e. flg should be passed as nil (flg=t is for internal use by the editor).

(editfindp x pat flg)

NOTE: Allows a program to use the edit find command as a pure predicate from outside the editor. x is an expression, pat a pattern. The value of editfindp is t if the command f pat would succeed, nil otherwise. editfindp calls editfpat to convert pat to the form expected by edit4e, unless flg=t. if the program is applying editfindp to several different expressions using the same pattern, it will be more efficient to call editfpat once, and then call editfindp with the converted pattern and flg=t.

(## g_com1 ...)

RETURNS: what the current expression would be after executing the edit commands com1 ... starting from the present edit chain. generates an error if any of comi cause errors. The current edit chain is never changed. example: (i r (quote x) (## (cons ..z))) replaces all x's in the current expression by the first cons containing a z.

CHAPTER 17

Hash Tables

17.1. Overview

A hash table is an object that can efficiently map a given object to another. Each hash table is a collection of entries, each of which associates a unique *key* with a *value*. There are elemental functions to add, delete, and find entries based on a particular key. Finding a value in a hash table is relatively fast compared to looking up values in, for example, an assoc list or property list.

Adding a key to a hash table modifies the hash table, and so it is a descructive operation.

There are two different kinds of hash tables: those that use the function *equal* for the comparing of keys, and those that use *eq*, the default. If a key is "eq" to another object, then a match is assumed. Likewise with "equal".

17.2. Functions

(makeht 'x_size ['s_test])

RETURNS: A hash table of x_size hash buckets. If present, s_test is used as the test to compare keys in the hash table, the default being **eq**. *Equal* might be used to create a hash table where the keys are to be lists (or any lisp object).

NOTE: At this time, hash tables are implemented on top of vectors.

(hash-table-p 'H_arg)

RETURNS: t if H_arg is a hash table.

NOTE: Since hash tables are really vectors, the lisp type of a hash table is a vector, so that given a vector, this function will return t.

(gethash 'g_key 'H_htable ['g_default])

RETURNS: the value associated the key g_key in hash table H_htable. If there is not an entry given by the key and g_default is specified, then g_default is returned, otherwise, a symbol that is unbound is returned. This is so that **nil** can be a associated with a key.

NOTE: *setf* may be used to set the value assocaited with a key.

(remhash 'g_key 'H_htable**)**

RETURNS:t if there was an entry for g_key in the hash table H_htable, nil otherwise. In the case of a match, the entry and associated object are removed from the hash table.

(maphash 'u_func 'H_htable**)**

RETURNS:nil.

NOTE: The function u_func is applied to every element in the hash table H_htable. The function is called with two arguments: the key and value of an element. The mapped function should not add or delete object from the table because the results would be unpredicable.

(clrhash 'H_htable**)**

RETURNS:the hash table cleared of all entries.

(hash-table-count 'H_htable**)**

RETURNS:the number of entries in H_htable. Given a new hash table with no entries, this function returns zero.

```
; make a vanilla hash table using "eq" to compare items...
-> (setq black-box (makeht 20))
hash-table[26]
-> (hash-table-p black-box)
t
-> (hash-table-count black-box)
0
-> (setf (gethash 'key black-box) '(the value associated with the key))
key
-> (gethash 'key black-box)
(the value associated with the key)
-> (hash-table-count black-box)
1
-> (addhash 'composer black-box 'franz)
composer
-> (gethash 'composer black-box)
franz
-> (maphash '(lambda (key val) (msg "key " key " value " val N)) black-box)
key composer value franz
key key value (the value associated with the key)
nil
-> (clrhash black-box)
hash-table[26]
-> (hash-table-count black-box)
0
-> (maphash '(lambda (key val) (msg "key " key " value " val N)) black-box)
nil

; here is an example using "equal" as the comparator
-> (setq ht (makeht 10 'equal))
hash-table[16]
-> (setf (gethash '(this is a key) ht) '(and this is the value))
(this is a key)
-> (gethash '(this is a key) ht)
(and this is the value)
; the reader makes a new list each time you type it...
-> (setq x '(this is a key))
(this is a key)
-> (setq y '(this is a key))
(this is a key)
; so these two lists are really different lists that compare "equal"
; not "eq"
-> (eq x y)
nil
; but since we are using "equal" to compare keys, we are OK...
-> (gethash x ht)
(and this is the value)
-> (gethash y ht)
(and this is the value)
```

APPENDIX A

Index to FRANZ LISP Functions

APPENDIX B

Special Symbols

The values of these symbols have a predefined meaning. Some values are counters while others are simply flags whose value the user can change to affect the operation of lisp system. In all cases, only the value cell of the symbol is important, the function cell is not. The value of some of the symbols (like **ER%misc**) are functions - what this means is that the value cell of those symbols either contains a lambda expression, a binary object, or symbol with a function binding.

The values of the special symbols are:

$gccount$ – The number of garbage collections which have occurred.

$gcprint – If bound to a non nil value, then after each garbage collection and subsequent storage allocation a summary of storage allocation will be printed.

$ldprint – If bound to a non nil value, then during each *fasl* or *cfasl* a diagnostic message will be printed.

ER%all – The function which is the error handler for all errors (see §10)

ER%brk – The function which is the handler for the error signal generated by the evaluation of the *break* function (see §10).

ER%err – The function which is the handler for the error signal generated by the evaluation of the *err* function (see §10).

ER%misc – The function which is the handler of the error signal generated by one of the unclassified errors (see §10). Most errors are unclassified at this point.

ER%tpl – The function which is the handler to be called when an error has occurred which has not been handled (see §10).

ER%undef – The function which is the handler for the error signal generated when a call to an undefined function is made.

^w – When bound to a non nil value this will prevent output to the standard output port (poport) from reaching the standard output (usually a terminal). Note that ^w is a two character symbol and should not be confused with ^W which is how we would denote control-w. The value of ^w is checked when the standard output buffer is flushed which occurs after a *terpr*, *drain* or when the buffer overflows. This is most useful in conjunction with ptport described below. System error handlers rebind ^w to nil when they are invoked to assure that error messages are not lost. (This was introduced for Maclisp compatibility).

defmacro-for-compiling – The has an effect during compilation. If non-nil it causes macros defined by defmacro to be compiled and included in the object file.

environment – The UNIX environment in assoc list form.

errlist – When a *reset* is done, the value of errlist is saved away and control is thrown to the top level. *Eval* is then mapped over the saved away value of this list.

errport – This port is initially bound to the standard error file.

evalhook – The value of this symbol, if bound, is the name of a function to handle evalhook traps (see §14.4)

float-format – The value of this symbol is a string which is the format to be used by print to print flonums. See the documentation on the UNIX function printf for a list of allowable formats.

funcallhook – The value of this symbol, if bound, is the name of a function to handle funcallhook traps (see §14.4).

gcdisable – If non nil, then garbage collections will not be done automatically when a collectable data type runs out.

ibase – This is the input radix used by the lisp reader. It may be either eight or ten. Numbers followed by a decimal point are assumed to be decimal regardless of what ibase is.

linel – The line length used by the pretty printer, pp. This should be used by *print* but it is not at this time.

nil – This symbol represents the null list and thus can be written (). Its value is always nil. Any attempt to change the value will result in an error.

piport – Initially bound to the standard input (usually the keyboard). A read with no arguments reads from piport.

poport – Initially bound to the standard output (usually the terminal console). A print with nó second argument writes to poport. See also: ^w and ptport.

prinlength – If this is a positive fixnum, then the *print* function will print no more than prinlength elements of a list or hunk and further elements abbreviated as '...'. The initial value of prinlength is nil.

prinlevel – If this is a positive fixnum, then the *print* function will print only prinlevel levels of nested lists or hunks. Lists below this level will be abbreviated by '&' and hunks below this level will be abbreviated by a '%'. The initial value of prinlevel is nil.

ptport – Initially bound to nil. If bound to a port, then all output sent to the standard output will also be sent to this port as long as this port is not also the standard output (as this would cause a loop). Note that ptport will not get a copy of whatever is sent to poport if poport is not bound to the standard output.

readtable – The value of this is the current readtable. It is an array but you should NOT try to change the value of the elements of the array using the array functions. This is because the readtable is an array of bytes and the smallest unit the array functions work with is a full word (4 bytes). You can use *setsyntax* to change the values and *(status syntax ...)* to read the values.

t – This symbol always has the value t. It is possible to change the value of this symbol for short periods of time but you are strongly advised against it.

top-level – In a lisp system without /usr/lib/lisp/toplevel.l loaded, after a *reset* is done, the lisp system will *funcall* the value of top-level if it is non nil. This provides a way for the user to introduce his own top level interpreter. When /usr/lib/lisp/toplevel.l is loaded, it sets top-level to franz-top-level and changes the *reset* function so that once franz-top-level starts, it cannot be replaced by changing top-level. Franz-top-level does provide a way of changing the top level however, and that is through user-top-level.

user-top-level – If this is bound then after a *reset*, the top level function will *funcall* the value of this symbol rather than go through a read eval print loop.

APPENDIX C

Short Subjects.

The Garbage Collector

The garbage collector is invoked automatically whenever a collectable data type runs out. All data types are collectable except strings and atoms are not. After a garbage collection finishes, the collector will call the function *gcafter* which should be a lambda of one argument. The argument passed to *gcafter* is the name of the data type which ran out and caused the garbage collection. It is *gcafter*'s responsibility to allocate more pages of free space. The default *gcafter* makes its decision based on the percentage of space still in use after the garbage collection. If there is a large percentage of space still in use, *gcafter* allocates a larger amount of free space than if only a small percentage of space is still in use. The default *gcafter* will also print a summary of the space in use if the variable *$gcprint* is non nil. The summary always includes the state of the list and fixnum space and will include another type if it caused the garbage collection. The type which caused the garbage collection is preceded by an asterisk.

Debugging

There are two simple functions to help you debug your programs: *baktrace* and *showstack*. When an error occurs (or when you type the interrupt character), you will be left at a break level with the state of the computation frozen in the stack. At this point, calling the function *showstack* will cause the contents of the lisp evaluation stack to be printed in reverse chronological order (most recent first). When the programs you are running are interpreted or traced, the output of *showstack* can be very verbose. The function *baktrace* prints a summary of what *showstack* prints. That is, if showstack would print a list, *baktrace* would only print the first element of the list. If you are running compiled code with the *(status translink)* non nil, then fast links are being made. In this case, there is not enough information on the stack for *showstack* and *baktrace*. Thus, if you are debugging compiled code you should probably do *(sstatus translink nil)*.

If the contents of the stack don't tell you enough about your problem, the next thing you may want to try is to run your program with certain functions traced. You can direct the trace package to stop program execution when it enters a function, allowing you to examine the contents of variables or call other functions. The trace package is documented in Chapter 11.

It is also possible to single step the evaluator and to look at stack frames within lisp. The programs which perform these actions are described in Chapters 14 and 15.

The Interpreter's Top Level

The default top level interpreter for Franz, named *franz-top-level* is defined in /usr/lib/lisp/toplevel.l It is given control when the lisp system starts up because the variable top-level is bound to the symbol *franz-top-level.* The first action *franz-top-level* takes is to print out the name of the current version of the lisp system. Then it loads the file .lisprc from the HOME directory of the person invoking the lisp system if that file exists. The .lisprc file allows you to set up your own defaults, read in files, set up autoloading or anything else you might want to do to personalize the lisp system. Next, the top level goes into a prompt-read-eval-print loop. Each time around the loop, before printing the prompt it checks if the variable user-top-level is bound. If so, then the value of user-top-level will be *funcall*ed. This provides a convenient way for a user to introduce his own top level (Liszt, the lisp compiler, is an example of a program which uses this). If the user types a ^D (which is the end of file character), and the standard input is not from a keyboard, the lisp system will exit. If the standard input is a keyboard and if the value of *(status ignoreeof)* is nil, the lisp system will also exit. Otherwise the end of file will be ignored. When a *reset* is done the current value of *errlist* is saved away and control is thrown back up to the top level where *eval* is mapped over the saved value of *errlist.*

INGRES

VERSION 8

REFERENCE MANUAL

5/12/86

by

Joe Kalash
Lisa Rodgin
Zelaine Fong
Jeff Anton

ACKNOWLEDGEMENTS

We would like to acknowledge the people who have worked on INGRES in the past:

Eric Allman
Rick Birman
Bob Epstein
James Ford
Paula Hawthorn
Gerald Held
Peter Kreps
Marc Meyer
Jeff Ranstrom
Dan Ries
Peter Rubinstein
Polly Siegal
Mike Ubell
John Woodfill
Nick Whyte
Karel Youssefi
William Zook

FOOTNOTE

UNIX is a trademark of Bell Laboratories.

APPEND(QUEL) – append tuples to a relation
append [to] relname (target_list) [**where** qual]

COPY(QUEL) – copy data into/from a relation from/into a UNIX file.
copy relname (domname = format {, domname = format })
direction "filename"

CREATE(QUEL) – create a new relation
create relname (domname*1* = format {, domname*2* = format })

DEFINE(QUEL) – define subschema
define view name (target list) [**where** qual]
define permit oplist { **on** | **of** | **to** } var [(attlist)] **to** name [**at** term] [**from** time **to** time] [**on** day **to** day] [**where** qual]
define integrity on var **is** qual

DELETE(QUEL) – delete tuples from a relation
delete tuple_variable [**where** qual]

DELIM(QUEL) – specify a name for a pattern of characters
destroy delim groupname (delimitor, pattern)
pattern = ["[character list]" | "{character list}"]*

DESTROY(QUEL) – destroy existing relation(s)
destroy relname { , relname }
destroy [**permit** | **integrity**] relname [integer { , integer } | **all**]
destroy delim groupname

HELP(QUEL) – get information about how to use INGRES, about relations in the database,
help [relname] ["section"] {, relname}{, "section"}
help view relname {, relname}
help permit relname {, relname}
help integrity relname {, relname}
help delim [groupname] {, groupname}

INDEX(QUEL) – create a secondary index on an existing relation.
index on relname **is** indexname (domain*1* { ,domain*2*})

INTEGRITY(QUEL) – define integrity constraints
define integrity on var **is** qual

MACROS(QUEL) – terminal monitor macro facility

MODIFY(QUEL) – convert the storage structure of a relation
modify relname **to** storage-structure [**on** key1 [: *sortorder*] [{ , key2 [: *sortorder*] }]] [**where** [**fillfactor** = *n*] [, **minpages** = *n*] [, **maxpages** = *n*] [, **lidn** = *lidname*] [

MONITOR(QUEL) – interactive terminal monitor

ORDERED(QUEL) – storage structure type

PERMIT(QUEL) – add permissions to a relation
define permit oplist { **on** | **of** | **to** } var [(attlist)]
to name [**at** term] [**from** time **to** time]
[**on** day **to** day] [**where** qual]

PRINT(QUEL) – print relation(s)
print relname {, relname}

QUEL(QUEL) – **QUE**ry **L**anguage for INGRES

RANGE(QUEL) – declare a variable to range over a relation
range of variable **is** relname

REPLACE(QUEL) – replace values of domains in a relation
replace tuple_variable (target_list) [**where** qual]

This manual is a reference manual for the INGRES data base system. It documents the use of INGRES in a very terse manner. To learn how to use INGRES, refer to the document called "A Tutorial on INGRES".

The INGRES reference manual is subdivided into four parts:

Quel	describes the commands and features which are used inside of INGRES.
Unix	describes the INGRES programs which are executable as UNIX commands.
Files	describes some of the important files used by INGRES.
Error	lists all the user generatable error messages along with some elaboration as to what they mean or what we think they mean.

Each entry in this manual has one or more of the following sections:

NAME section
: This section repeats the name of the entry and gives an indication of its purpose.

SYNOPSIS section
: This section indicates the form of the command (statement). The conventions which are used are as follows:

  Bold face names are used to indicate reserved keywords.
  Lower case words indicate generic types of information which must be supplied by the user; legal values for these names are described in the DESCRIPTION section.
  Square brakets ([]) indicate that the enclosed item is optional.
  Braces ({}) indicate an optional item which may be repeated. In some cases they indicate simple (non-repeated) grouping; the usage should be clear from context.

  When these conventions are insufficient to fully specify the legal format of a command a more general form is given and the allowable subsets are specified in the DESCRIPTION section.

DESCRIPTION section
: This section gives a detailed description of the entry with references to the generic names used in the SYNOPSIS section.

EXAMPLE section
: This section gives one or more examples of the use of the entry. Most of these examples are based on the following relations:

```
emp(name,sal,mgr,bdate)
and
newemp(name,sal,age)
and
parts(pnum, pname, color, weight, qoh)
```

SEE ALSO section
: This section gives the names of entries in the manual which are closely related to the current entry or which are referenced in the description of the current entry.

BUGS section
: This section indicates known bugs or deficiencies in the command.

To start using INGRES you must be entered as an INGRES user; this is done by the INGRES administrator who will enter you in the "users" file (see users(files)). To start using ingres see the section on ingres(unix), quel(quel), and monitor(quel).

NAME

append – append tuples to a relation

SYNOPSIS

append [to] relname (target_list) [**where** qual]

DESCRIPTION

Append adds tuples which satisfy the qualification to *relname*. *Relname* must be the name of an existing relation. The *target_list* specifies the values of the attributes to be appended to *relname*. The domains may be listed in any order. Attributes of the result relation which do not appear in the *target_list* as result_attnames (either explicitly or by default) are assigned default values of 0, for numeric attributes, or blank, for character attributes.

Values or expressions of any numeric type may be used to set the value of a numeric type domain. Conversion to the result domain type takes place. Numeric values cannot be directly assigned to character domains. Conversion from numeric to character can be done using the **ascii** operator (see quel(quel)). Character values cannot be directly assigned to numeric domains. Use the **int1**, **int2**, etc. functions to convert character values to numeric (see quel(quel)).

The keyword **all** can be used when it is desired to append all domains of a relation.

An *append* may only be issued by the owner of the relation or a user with *append* permission on the given relation.

EXAMPLE

```
/* Make new employee Jones work for Smith */
  range of n is newemp
  append to emp(n.name, n.sal, mgr = "Smith", bdate = 1975-n.age)
                where n.name = "Jones"
/* Append the newemp1 relation to newemp */
  range of n1 is newemp1
  append to newemp(n1.all)
```

SEE ALSO

copy(quel), permit(quel), quel(quel), retrieve(quel)

DIAGNOSTICS

Use of a numeric type expression to set a character type domain or vice versa will produce diagnostics.

BUGS

Duplicate tuples appended to a relation stored as a "paged heap" (unkeyed, unstructured) are not removed.

NAME

copy – copy data into/from a relation from/into a UNIX file.

SYNOPSIS

copy relname (domname = format {, domname = format })
direction "filename"

DESCRIPTION

Copy moves data between INGRES relations and standard UNIX files. *Relname* is the name of an existing relation. In general *domname* identifies a domain in *relname*. *Format* indicates the format the UNIX file should have for the corresponding domain. *Direction* is either **into** or **from.** *Filename* is the full UNIX pathname of the file.

On a copy **from** a file to a relation, the relation cannot have a secondary index, it must be owned by you, and it must be updatable (not a secondary index or system relation).

Copy cannot be used on a relation which is a view. For a copy **into** a UNIX file, you must either be the owner of the relation or the relation must have retrieve permission for all users, or all permissions for all users.

The formats allowed by copy are:

i1,i2,i4 – The data is stored as an integer of length 1, 2, or 4 bytes in the UNIX file.

f4,f8 – The data is stored as a floating point number (either single or double precision) in the UNIX file.

c1,c2,...,c255 – The data is stored as a fixed length string of characters.

c0 – Variable length character string.

d0,d1,...,d255 – Dummy domain.

Corresponding domains in the relation and the UNIX file do not have to be the same type or length. *Copy* will convert as necessary. When converting anything except character to character, *copy* checks for overflow. When converting from character to character, *copy* will blank pad or truncate on the right as necessary.

The domains should be ordered according to the way they should appear in the UNIX file. Domains are matched according to name, thus the order of the domains in the relation and in the UNIX file does not have to be the same.

Copy also provides for variable length strings and dummy domains. The action taken depends on whether it is a copy **into** or a copy **from.** Delimitors for variable length strings and for dummy domains can be selected from the list of:

nl – new line character
tab – tab character
sp – space
nul or **null** – null character
comma – comma
colon – colon
dash – dash
lparen – left parenthesis
rparen – right parenthesis
x – any single character 'x'

The special meaning of any delimitor can be turned off by preceeding the delimitor with a '\'. The type specifier can optionally be in quotes ("c0delim"). This is usefully if you wish to use a single character delimitor which has special meaning to the QUEL parser.

When the *direction* is from, *copy* appends data into the relation **from** the UNIX file. Domains in the INGRES relation which are not assigned values from the UNIX file are assigned the default value of zero for numeric domains, and blank for character domains. When copying in this direction the following special meanings apply:

c0*delim* – The data in the UNIX file is a variable length character string terminated by the delimitor *delim*. If *delim* is missing then the first comma, tab, or newline encountered will terminate the string. The delimitor is not copied.

For example:

pnum=c0 – string ending in comma, tab, or nl.
pnum=c0nl – string ending in nl.
pnum=c0sp – string ending in space.
pnum=c0z – string ending in the character 'z'.
pnum="c0%" – string ending in the character '%'.

A delimitor can be escaped by preceeding it with a '\'. For example, using **name = c0**, the string "Blow\, Joe," will be accepted into the domain as "Blow, Joe".

d0*delim* – The data in the UNIX file is a variable length character string delimited by *delim*. The string is read and discarded. The delimitor rules are identical for **c0** and **d0**. The domain name is ignored.

d1,d2,...,d255 – The data in the UNIX file is a fixed length character string. The string is read and discarded. The domain name is ignored.

When the direction is **into,** *copy* transfers data **into** the UNIX file from the relation. If the file already existed, it is truncated to zero length before copying begins. When copying in this direction, the following special meanings apply:

c0 – The domain value is converted to a fixed length character string and writted into the UNIX file. For character domains, the length will be the same as the domain length. For numeric domains, the standard INGRES conversions will take place as specified by the '–i', '–f', and '–c' flags (see ingres(unix)).

c0*delim* – The domain will be converted according to the rules for **c0** above. The one character delimitor will be inserted immediately after the domain.

d1,d2,...,d255 – The domain name is taken to be the name of the delimitor. It is written into the UNIX file 1 time for **d1,** 2 times for **d2,** etc.

d0 – This format is ignored on a copy **into.**

d0*delim* – The *delim* is written into the file. The domain name is ignored.

If no domains appear in the copy command (i.e. copy relname () into/from "filename") then *copy* automatically does a "bulk" copy of all domains, using the order and format of the domains in the relation. This is provided as a convenient shorthand notation for copying and restoring entire relations.

To *copy* into a relation, you must be the owner or all users must have all permissions set. Correspondingly, to *copy* from a relation you must own the relation or all users must have at least retrieve permission on the relation. Also, you may not *copy* a view.

EXAMPLE

```
/* Copy data into the emp relation */
  copy emp (name=c10,sal=f4,bdate=i2,mgr=c10,xxx=d1)
          from "/mnt/me/myfile"

/* Copy employee names and their salaries into a file */
  copy emp (name=c0,comma=d1,sal=c0,nl=d1)
           into "/mnt/you/yourfile"

/* Bulk copy employee relation into file */
  copy emp ()
           into "/mnt/ours/ourfile"
```

```
/* Bulk copy employee relation from file */
  copy emp ()
          from "/mnt/thy/thyfile"
```

SEE ALSO

append(quel), create(quel), quel(quel), permit(quel), view(quel), ingres(unix)

BUGS

Copy stops operation at the first error.

When specifying *filename*, the entire UNIX directory pathname must be provided, since INGRES operates out of a different directory than the user's working directory at the time INGRES is invoked.

NAME

create – create a new relation

SYNOPSIS

create relname (domname*1* = format {, domname*2* = format })

DESCRIPTION

Create will enter a new relation into the data base. The relation will be "owned" by the user and will be set to expire after seven days. The name of the relation is *relname* and the domains are named *domname1, domname2,* etc. The domains are created with the type specified by *format.* Formats are described in the quel(quel) manual section.

The relation is created as a paged heap with no data initially in it.

A relation can have no more than 49 domains. A relation cannot have the same name as a system relation.

EXAMPLE

```
/* Create relation emp with domains name, sal and bdate */
  create emp (name = c10, salary = f4, bdate = i2)
```

SEE ALSO

append(quel), copy(quel), destroy(quel), save(quel)

BUGS

NAME

define – define subschema

SYNOPSIS

define view name (target list) [**where** qual]
define permit oplist { **on** | **of** | **to** } var [(attlist)] **to** name [**at** term] [**from** time **to** time] [**on** day **to** day] [**where** qual]
define integrity on var **is** qual

DESCRIPTION

The *define* statement creates entries for the subschema definitions. See the manual sections listed below for complete descriptions of these commands.

SEE ALSO

integrity(quel), permit(quel), view(quel)

NAME

delete – delete tuples from a relation

SYNOPSIS

delete tuple_variable [**where** qual]

DESCRIPTION

Delete removes tuples which satisfy the qualification *qual* from the relation that they belong to. The *tuple_variable* must have been declared to range over an existing relation in a previous *range* statement. *Delete* does not have a *target_list.* The *delete* command requires a tuple variable from a *range* statement, and not the actual relation name. If the qualification is not given, the effect is to delete all tuples in the relation. The result is a valid, but empty relation.

To *delete* tuples from a relation, you must be the owner of the relation, or have *delete* permission on the relation.

EXAMPLE

```
/* Remove all employees who make over $30,000 */
  range of e is emp
  delete e where e.sal > 30000
```

SEE ALSO

destroy(quel), permit(quel), quel(quel), range(quel)

BUGS

NAME

delim - specify a name for a pattern of characters

SYNOPSIS

destroy delim groupname (delimitor, pattern)
pattern = ["[character list]" | "{character list}"]*

DESCRIPTION

The delim statement allows the user to specify a name for a certain pattern of characters, which may be used with the substring facility. The delimitors are stored in groups, which may be used and unused together.

The pattern for a delimitor is specified using a modified version of BNF grammar. A character list is either a list of the characters to match, or a range of characters separated by a "-", or a combination or both of the above. A character list in brackets indicates that one of the characters must match exactly once, and a character list in braces indicates that one of the characters may match zero or more times. The pattern is composed of a list of character lists.

After a delimitor has been defined, to be used in an ingres query it must be activated using the 'use' command.

EXAMPLE

```
/* will match a sequence of alphabetic characters beginning
with a capital letter */
define delim paper(word, "[A-Z]{a-z}")

/* will match a decimal number */
define delim math (dec, "[0-9]{0-9}[.][0-9]{0-9}")
```

SEE ALSO

destroy(quel), quel(quel), use(quel)

NAME

destroy – destroy existing relation(s)

SYNOPSIS

destroy relname { , relname }
destroy [**permit** | **integrity**] relname [integer { , integer } | **all**]
destroy delim groupname

DESCRIPTION

Destroy removes relations from the data base, removes constraints or permissions from a relation, and removes user-defined delimitors. Only the relation owner may destroy a relation or its permissions and integrity constraints. A relation may be emptied of tuples, but not destroyed, using the delete statement or the modify statement.

If the relation being destroyed has secondary indices on it, the secondary indices are also destroyed. Destruction of just a secondary index does not affect the primary relation it indexes.

To destroy individual permissions or constraints for a relation, the *integer* arguments should be those printed by a **help permit** (for **destroy permit**) or a **help integrity** (for **destroy integrity**) on the same relation. To destroy all constraints or permissions, the **all** keyword may be used in place of individual integers. To destroy constraints or permissions, either the *integer* arguments or the **all** keyword must be present.

To destroy a delimitor group, the groupname must be specified. This destroys the delimitors permanently, as opposed to unusing the group.

EXAMPLE

```
/* Destroy the emp relation */
   destroy emp
   destroy emp, parts

/* Destroy some permissions on parts, and all integrity
 * constraints on employee
 */
   destroy permit parts 0, 4, 5
   destroy integrity employee

/* Destroy the "paper" delimitor group */
   destroy delim paper
```

SEE ALSO

create(quel), delete(quel), delim(quel), help(quel), index(quel), modify(quel)

NAME

help – get information about how to use INGRES, about relations in the database, or about user-defined delimitors.

SYNOPSIS

help [relname] ["section"] {, relname}{, "section"}
help view relname {, relname}
help permit relname {, relname}
help integrity relname {, relname}
help delim [groupname] {, groupname}

DESCRIPTION

Help may be used to obtain sections of this manual, information on the content of the current data base, information about specific relations in the data base, view definitions, or protection and integrity constraints on a relation. The legal forms are as follow:

help "*section* " – Produces a copy of the specified section of the INGRES Reference Manual, and prints it on the standard output device.
help – Gives information about all relations that exist in the current database.
help *relname* {, *relname*} – Gives information about the specified relations.
help " " – Gives the table of contents.
help view *relname* {, *relname*} – Prints view definitions of specified views.
help permit *relname* {, *relname*} – Prints permissions on specified relations.
help integrity *relname* {, *relname*} – Prints integrity constraints on specified relations.
help delim – Prints a list of all the delimitor groups defined.
help delim *groupname* {, *groupname*} – Prints a list of the delimitors in each group given, and the patterns which they represent.

The **permit** and **integrity** forms print out unique identifiers for each constraint. These identifiers may be used to remove the constraints with the *destroy* statement.

EXAMPLE

```
help
help "help"  /* prints this page of the manual */
help quel
help emp
help emp, parts, "help", supply
help view overp_view
help permit parts, employee
help integrity parts, employee
help delim
help delim paper, math
```

SEE ALSO

destroy(quel)

BUGS

Alphabetics appearing within the section name must be in lower-case to be recognized.

NAME

index – create a secondary index on an existing relation.

SYNOPSIS

index on relname **is** indexname (domain*1* { ,domain*2*})

DESCRIPTION

Index is used to create secondary indices on existing relations in order to make retrieval and update with secondary keys more efficient. The secondary key is constructed from relname domains 1, 2,...,6 in the order given. Only the owner of a relation is allowed to create secondary indices on that relation.

In order to maintain the integrity of the index, users will NOT be allowed to directly update secondary indices. However, whenever a primary relation is changed, its secondary indices will be automatically updated by the system. Secondary indices may be modified to further increase the access efficiency of the primary relation. When an index is first created, it is automatically modified to an isam storage structure on all its domains. If this structure is undesirable, the user may override the default isam structure by using the **-n** switch (see ingres(unix)), or by entering a *modify* command directly.

If a *modify* or *destroy* command is used on *relname*, all secondary indices on *relname* are destroyed.

Secondary indices on other indices, or on system relations are forbidden.

EXAMPLE

```
/* Create a secondary index called "x" on relation "emp" */
  index on emp is x(mgr,sal)
```

SEE ALSO

copy(quel), destroy(quel), modify(quel)

BUGS

At most 6 domains may appear in the key.

The *copy* command cannot be used to copy into a relation which has secondary indices.

The default structure isam is a poor choice for an index unless the range of retrieval is small.

NAME

integrity – define integrity constraints

SYNOPSIS

define integrity on var **is** qual

DESCRIPTION

The *integrity* statement adds an integrity constraint for the relation specified by *var*. After the constraint is placed, all updates to the relation must satisfy *qual*. *Qual* must be true when the *integrity* statement is issued or else a diagnostic is issued and the statement is rejected.

In the current implementation, *integrity* constraints are not flagged – bad updates are simply (and silently) not performed.

Qual must be a single variable qualification and may not contain any aggregates.

integrity statement may be issued only by the relation owner.

EXAMPLE

```
/* Ensure all employees have positive salaries */
  range of e is employee
  define integrity on e is e.salary > 0
```

SEE ALSO

destroy(quel)

NAME

macros – terminal monitor macro facility

DESCRIPTION

The terminal monitor macro facility provides the ability to tailor the QUEL language to the user's tastes. The macro facility allows strings of text to be removed from the query stream and replaced with other text. Also, some built in macros change the environment upon execution.

Basic Concepts

All macros are composed of two parts, the *template* part and the *replacement* part. The template part defines when the macro should be invoked. For example, the template "ret" causes the corresponding macro to be invoked upon encountering the word "ret" in the input stream. When a macro is encountered, the template part is removed and replaced with the replacement part. For example, if the replacement part of the "ret" macro was "retrieve", then all instances of the word "ret" in the input text would be replaced with the word "retrieve", as in the statement

```
ret (p.all)
```

Macros may have parameters, indicated by a dollar sign. For example, the template "get $1" causes the macro to be triggered by the word "get" followed by any other word. The word following "get" is remembered for later use. For example, if the replacement part of the "get" macro where

```
retrieve (p.all) where p.pnum = $1
```

then typing "get 35" would retrieve all information about part number 35.

Defining Macros

Macros can be defined using the special macro called "define". The template for the define macro is (roughly)

```
{define; $t; $r}
```

where $t and $r are the template and replacement parts of the macro, respectively.

Let's look at a few examples. To define the "ret" macro discussed above, we would type:

```
{define; ret; retrieve}
```

When this is read, the macro processor removes everything between the curly braces and updates some tables so that "ret" will be recognized and replaced with the word "retrieve". The define macro has the null string as replacement text, so that this macro seems to disappear.

A useful macro is one which shortens range statements. It can be defined with

```
{define; rg $v $r; range of $v is $r}
```

This macro causes the word "rg" followed by the next two words to be removed and replaced by the words "range of", followed by the first word which followed "rg", followed by the word "is", followed by the second word which followed "rg". For example, the input

```
rg p parts
```

becomes the same as

```
range of p is parts
```

Evaluation Times

When you type in a define statement, it is not processed immediately, just as queries are saved rather than executed. No macro processing is done until the query buffer is evaluated. The commands \go, \list, and \eval evaluate the query buffer. \go sends the results to INGRES, \list prints them on your terminal, and \eval puts the result back into the query buffer.

It is important to evaluate any define statements, or it will be exactly like you did not type them in at all. A common way to define macros is to type

```
{define . . . }
\eval
\reset
```

If the \eval was left out, there is no effect at all.

Quoting

Sometimes strings must be passed through the macro processor without being processed. In such cases the grave and acute accent marks (` and ´) can be used to surround the literal text. For example, to pass the word "ret" through without converting it to "retrieve" we could type

```
`ret´
```

Another use for quoting is during parameter collection. If we want to enter more than one word where only one was expected, we can surround the parameter with accents.

The backslash character quotes only the next character (like surrounding the character with accents). In particular, a grave accent can be used literally by preceeding it with a backslash.

Since macros can normally only be on one line, it is frequently useful to use a backslash at the end of the line to hide the newline. For example, to enter the long "get" macro, you might type:

```
{define; get $n; retrieve (e.all) \
where e.name = "$n"}
```

The backslash always quotes the next character even when it is a backslash. So, to get a real backslash, use two backslashes.

More Parameters

Parameters need not be limited to the word following. For example, in the template descriptor for define:

```
{define; $t; $r}
```

the $t parameter ends at the first semicolon and the $r parameters ends at the first right curly brace. The rule is that the character which follows the parameter specifier terminates the parameter; if this character is a space, tab, newline, or the end of the template then one word is collected.

As with all good rules, this one has an exception. Since system macros are always surrounded by curly braces, the macro processor knows that they must be properly nested. Thus, in the statement

```
{define; x; {sysfn}}
```

The first right curly brace will close the "sysfn" rather than the "define". Otherwise this would have to be typed

```
{define; x; `{sysfn}´}
```

Words are defined in the usual way, as strings of letters and digits plus the underscore character.

Other Builtin Macros

There are several other macros built in to the macro processor. In the following description, some of the parameter specifiers are marked with two dollar signs rather than one; this will be discussed in the section on prescanning below.

{define; $$t; $$r} defines a macro as discussed above. Special processing occurs on the template part which will be discussed in a later section.

{rawdefine; $$t; $$r} is another form of define, where the special processing does not take place.

{remove; $$n} removes the macro with name $n. It can remove more than one macro, since it actually removes all macros which might conflict with $n under some circumstance. For example, typing

```
{define; get part $n; . . . }
{define; get emp $x; . . . }
{remove; get}
```

would cause both the get macros to be removed. A call to

```
{remove; get part}
```

would have only removed the first macro.

{type $$s} types $s onto the terminal.

{read $$s} types $s and then reads a line from the terminal. The line which was typed replaces the macro. A macro called "{readcount}" is defined containing the number of characters read. A control-D (end of file) becomes −1, a single newline becomes zero, and so forth.

{readdefine; $$n; $$s} also types $s and reads a line, but puts the line into a macro named $n. The replacement text is the count of the number of characters in the line. {readcount} is still defined.

{ifsame; $$a; $$b; $t; $f} compares the strings $a and $b. If they match exactly then the replacement text becomes $t, otherwise it becomes $f.

{ifeq; $$a; $$b; $t; $f} is similar, but the comparison is numeric.

{ifgt; $$a; $$b; $t; $f} is like ifeq, but the test is for $a strictly greater than $b.

{substr; $$f; $$t; $$s} returns the part of $s between character positions $f and $t, numbered from one. If $f or $t are out of range, they are moved in range as much as possible.

{dump; $$n} returns the value of the macro (or macros) which match $n (using the same algorithm as remove). The output is a rawdefine statement so that it can be read back in. {dump} without arguments dumps all macros.

Metacharacters

Certain characters are used internally. Normally you will not even see them, but they can appear in the output of a dump command, and can sometimes be used to create very fancy macros.

\| matches any number of spaces, tabs, or newlines. It will even match zero, but only between words, as can occur with punctuation. For example, \| will match the spot between the last character of a word and a comma following it.

\^ matches exactly one space, tab, or newline.

\& matches exactly zero spaces, tabs, or newlines, but only between words.

The Define Process

When you define a macro using define, a lot of special processing happens. This processing is such that define is not functionally complete, but still adequate for most requirements. If more power is needed, rawdefine can be used; however, rawdefine is particularly difficult to use correctly, and should only be used by gurus.

In define, all sequences of spaces, tabs, and newlines in the template, as well as all "non-spaces" between words, are turned into a single \| character. If the template ends with a parameter, the \& character is added at the end.

If you want to match a real tab or newline, you can use \t or \n respectively. For example, a macro which reads an entire line and uses it as the name of an employee would be defined with

```
{define; get $n\n; \
  ret (e.all) where e.name = "$n"}
```

This macro might be used by typing

```
get *Stan*
```

to get all information about everyone with a name which included "Stan". By the way, notice that it is ok to nest the "ret" macro inside the "get" macro.

Parameter Prescan

Sometimes it is useful to macro process a parameter before using it in the replacement part. This is particularly important when using certain builtin macros.

For prescan to occur, two things must be true: first, the parameter must be specified in the template with two dollar signs instead of one, and second, the actual parameter must begin with an "at" sign ("@") (which is stripped off).

For an example of the use of prescan, see "Special Macros" below.

Special Macros

Some special macros are used by the terminal monitor to control the environment and return results to the user.

{begintrap} is executed at the beginning of a query.

{endtrap} is executed after the body of a query is passed to INGRES.

{continuetrap} is executed after the query completes. The difference between this and endtrap is that endtrap occurs after the query is submitted, but before the query executes, whereas continuetrap is executed after the query executes.

{editor} can be defined to be the pathname of an editor to use in the \edit command.

{shell} can be defined to be the pathname of a shell to use in the \shell command.

{tuplecount} is set after every query (but before continuetrap is sprung) to be the count of the number of tuples which satisfied the qualification of the query in a retrieve, or the number of tuples changed in an update. It is not set for DBU functions. If multiple queries are run at once, it is set to the number of tuples which satisfied the last query run.

For example, to print out the number of tuples touched automatically after each query, you could enter:

```
{define; {begintrap}; {remove; {tuplecount}}}
{define; {continuetrap}; \
  {ifsame; @{tuplecount}; {tuplecount};; \
    {type @{tuplecount} tuples touched}}}
```

SEE ALSO

monitor(quel)

NAME

modify – convert the storage structure of a relation

SYNOPSIS

modify relname **to** storage-structure [**on** key1 [: *sortorder*] [{ , key2 [: *sortorder*] }]] [
where [**fillfactor** = *n*] [, **minpages** = *n*] [, **maxpages** = *n*] [, **lidn** = *lidname*] [

DESCRIPTION

Relname is modified to the specified storage structure. Only the owner of a relation can modify that relation. This command is used to increase performance when using large or frequently referenced relations. The storage structures are specified as follows:

isam – indexed sequential storage structure
cisam – compressed isam
hash – random hah storage structure
chash – compressed hash
heap – unkeyed and unstructured
cheap – compressed heap
heapsort – heap with tuples sorted and duplicates removed
cheapsort – compressed heapsort
truncated – heap with all tuples deleted
orderedn – ordered relation where n is the ordering dimension

The paper "Creating and Maintaining a Database in INGRES" (ERL Memo M77–71) discusses how to select storage structures based on how the relation is used.

The current compression algorithm only suppresses trailing blanks in character fields. A more effective compression scheme may be possible; but tradeoffs between that and a larger and slower compression algorithm are not clear.

If the *on* phrase is omitted when modifying to isam, cisam, hash or chash, the relation will automatically be keyed on the first domain. When modifying to heap or cheap the *on* phrase must be omitted. When modifying to heapsort or cheapsort the *on* phrase is optional.

When a relation is being sorted (isam, cisam, heapsort and cheapsort), the primary sort keys will be those specified in the *on* phrase (if any). The first key after the *on* phrase will be the most significant sort key and each successive key specified will be the next most significant sort key. Any domains not specified in the *on* phrase will be used as least significant sort keys in domain number sequence.

When a relation is modified to heapsort or cheapsort, the *sortorder* can be specified to be **ascending** or **descending.** The default is always **ascending.** Each key given in the *on* phrase can be optionally modified to be:

key:descending

which will cause that key to be sorted in descending order. For completeness, **ascending** can be specified after the colon (':'), although this is unnecessary since it is the default. **Descending** can be abbreviated by a single '**d**' and, correspondingly, **ascending** can be abreviated by a single '**a**'.

When modifying to *orderedn,* up to n ordering keys can be specified using the *on* clause. Ordering keys are used to specify the ordering of tuples in the new relation. Changes on key field values indicate the incrementing of a lid value for the lid corresponding to the key change. If no ordering keys are specified, only the lid corresponding to the lowest lid level is incremented by one for every new tuple. In this case,the order of the tuples is determined by their sort order on file. However, note that ordering does not destroy any current storage structures on a relation (except secondary indices).

Lidn can only be specified if modifying to *orderedn.* Default values are **lid1** , **lid2** , and **lid3.**

Fillfactor specifies the percentage (from 1 to 100) of each primary data page that should be filled with tuples, under ideal conditions. *Fillfactor* may be used with isam, cisam, hash and chash. Care should be taken when using large fillfactors since a non-uniform distribution of

key values could cause overflow pages to be created, and thus degrade access performance for the relation.

Minpages specifies the minimum number of primary pages a hash or chash relation must have. *Maxpages* specifies the maximum number of primary pages a hash or chash relation may have. *Minpages* and *maxpages* must be at least one. If both **minpages** and **maxpages** are specified in a modify, **minpages** cannot exceed **maxpages.**

Default values for **fillfactor**, **minpages**, and **maxpages** are as follows:

	FILLFACTOR	*MINPAGES*	*MAXPAGES*
hash	50	10	no limit
chash	75	1	no limit
isam	80	NA	NA
cisam	100	NA	NA

EXAMPLES

```
/* modify the emp relation to an indexed
      sequential storage structure with
      "name" as the keyed domain */

modify emp to isam on name

/* if "name" is the first domain of the emp relation,
      the same result can be achieved by */

modify emp to isam

/* do the same modify but request a 60% occupancy
      on all primary pages */

modify emp to isam on name where fillfactor = 60

/* modify the supply relation to compressed hash
      storage structure with "num" and "quan"
      as keyed domains */

modify supply to chash on num, quan

/* now the same modify but also request 75% occupancy
      on all primary, a minimum of 7 primary pages
      pages and a maximum of 43 primary pages */

modify supply to chash on num, quan
      where fillfactor = 75, minpages = 7,
      maxpages = 43

/* again the same modify but only request a minimum
      of 16 primary pages */

modify supply to chash on num, quan
      where minpages = 16

/* modify parts to a heap storage structure */

modify parts to heap

/* modify parts to a heap again, but have tuples
      sorted on "pnum" domain and have any duplicate
      tuples removed */

modify parts to heapsort on pnum
```

```
/* modify employee in ascending order by manager,
        descending order by salary and have any
        duplicate tuples removed */

modify employee to heapsort on manager, salary:descending

/* ordered relation */

modify text to ordered1 on lid where lid1 = lidfield
```

SEE ALSO

sysmod(unix) ordered(quel)

NAME

monitor – interactive terminal monitor

DESCRIPTION

The interactive terminal monitor is the primary front end to INGRES. It provides the ability to formulate a query and review it before issuing it to INGRES. If changes must be made, one of the UNIX text editors may be called to edit the *query buffer*.

Messages and Prompts.

The terminal monitor gives a variety of messages to keep the user informed of the status of the monitor and the query buffer.

As the user logs in, a login message is printed. This typically tells the version number and the login time. It is followed by the dayfile, which gives information pertinant to users.

When INGRES is ready to accept input, the message "go" is printed. This means that the query buffer is empty. The message "continue" means that there is information in the query buffer. After a \go command the query buffer is automatically cleared if another query is typed in, unless a command which affects the query buffer is typed first. These commands are \append, \edit, \print, \list, \eval, and \go. For example, typing

```
help parts
\go
print parts
```

results in the query buffer containing

```
print parts
```

whereas

```
help parts
\go
\print
print parts
```

results in the query buffer containing

```
help parts
print parts
```

An asterisk is printed at the beginning of each line when the monitor is waiting for the user to type input.

Commands

There are a number of commands which may be entered by the user to affect the query buffer or the user's environment. They are all preceeded by a backslash ('\'), and all are executed immediately (rather than at execution time like queries).

Some commands may take a filename, which is defined as the first significant character after the end of the command until the end of the line. These commands may have no other commands on the line with them. Commands which do not take a filename may be stacked on the line; for example

```
\date\go\date
```

will give the time before and after execution of the current query buffer.

\r
\reset — Erase the entire query (reset the query buffer). The former contents of the buffer are irretrieveably lost.

\p
\print — Print the current query. The contents of the buffer are printed on the user's terminal.

\l
\list — Print the current query as it will appear after macro processing. Any side effects of macro processing, such as macro definition, will occur.

\eval — Macro process the query buffer and replace the query buffer with the result. This is just like \list except that the output is put into the query buffer instead of to the terminal.

\e
\ed
\edit
\editor — Enter the UNIX text editor (see ED in the UNIX Programmer's Manual); use the ED command 'w' followed by 'q' to return to the INGRES monitor. If a filename is given, the editor is called with that file instead of the query buffer. If the macro "{editor}" is defined, that macro is used as the pathname of an editor, otherwise "/bin/ed" is used. It is important that you do not use the "e" command inside the editor; if you do the (obscure) name of the query buffer will be forgotten.

\g
\go — Process the current query. The contents of the buffer are macro processed, transmitted to INGRES, and run.

\a
\append — Append to the query buffer. Typing \a after completion of a query will override the auto-clear feature and guarantees that the query buffer will not be reset.

\time
\date — Print out the current time of day.

\s
\sh
\shell — Escape to the UNIX shell. Typing a control-d will cause you to exit the shell and return to the INGRES monitor. If there is a filename specified, that filename is taken as a shell file which is run with the query buffer as the parameter "$1". If no filename is given, an interactive shell is forked. If the macro "{shell}" is defined, it is used as the pathname of a shell; otherwise, "/bin/sh" is used.

\q
\quit — Exit from INGRES.

\cd
\chdir — Change the working directory of the monitor to the named directory.

\i
\include
\read — Switch input to the named file. Backslash characters in the file will be processed as read.

\w
\write — Write the contents of the query buffer to the named file.

\branch — Transfer control within a \include file. See the section on branching below.

\mark — Set a label for \branch.

\<any other character> — Ignore any possible special meaning of character following '\'. This allows the '\' to be input as a literal character. (See also quel(quel) - strings). It is important to note that backslash escapes are sometimes eaten up by the macro processor also; in general, send two backslashes if you want a backslash sent (even this is too simplistic [sigh] – try to avoid using backslashes at all).

Macros

For simplicity, the macros are described in the section macros(quel).

Branching

The \branch and \mark commands permit arbitrary branching within a \include file (similar to the "goto" and ":" commands in the shell). \mark should be followed with a label. \branch should be followed with either a label, indicating unconditional branch, or an expres-

sion preceeded by a question mark, followed by a label, indicating a conditional branch. The branch is taken if the expression is greater than zero. For example,

```
\branch ?{tuplecount}<=0 notups
```

branches to label "notups" if the "{tuplecount}" macro is less than or equal to zero.

The expressions usable in \branch statements are somewhat restricted. The operators +, –, *, /, <=, >=, <, >, =, and != are all defined in the expected way. The left unary operator "!" can be used as to indicate logical negation. There may be no spaces in the expression, since a space terminates the expression. .

Initialization

At initialization (login) time a number of initializations take place. First, a macro called "{pathname}" is defined which expands to the pathname of the INGRES subtree (normally "/mnt/ingres"); it is used by system routines such as demodb. Second, the initialization file .../files/startup is read. This file is intended to define system-dependent parameters, such as the default editor and shell. Third, a user dependent initialization file, specified by a field in the users file, is read and executed. This is normally set to the file ".ingres" in the user's home directory. The startup file might be used to define certain macros, execute common range statements, and soforth. Finally, control is turned over to the user's terminal.

An interrupt while executing either of the initialization files restarts execution of that step.

Flags

Certain flags may be included on the command line to INGRES which affect the operation of the terminal monitor. The –a flag disables the autoclear function. This means that the query buffer will never be automatically cleared; equivalently, it is as though a \append command were inserted after every \go. Note that this means that the user must explicitly clear the query buffer using \reset after every query. The –d flag turns off the printing of the dayfile. The –s flag turns off printing of all messages (except errors) from the monitor, including the login and logout messages, the dayfile, and prompts. It is used for executing "canned queries", that is, queries redirected from files.

SEE ALSO

ingres(unix), quel(quel), macros(quel)

DIAGNOSTICS

go	You may begin a fresh query.
continue	The previous query is finished and you are back in the monitor.
Executing . . .	The query is being processed by INGRES.
>>ed	You have entered the UNIX text editor.
>>sh	You have escaped to the UNIX shell.

Funny character nnn converted to blank
INGRES maps non-printing ASCII characters into blanks; this message indicates that one such conversion has just been made.

INCOMPATIBILITIES

Note that the construct

```
\rprint parts
```

(intended to reset the query buffer and then enter "print parts") no longer works, since "rprint" appears to be one word.

NAME

ordered – storage structure type

DESCRIPTION

Ordered relations are a special type of storage structure in INGRES. They are created by using the modify relation to orderedn command where n indicates the ordering dimension. Ordering does not destroy existing storage structures on a relation. The resulting relation is the old relation with n 4-byte integer LID fields "attached" to the the end of the relation. These fields are different from conventional attribute fields because they can be dynamically adjusted by the system during updates to maintain a consistent ordering of tuples in a relation. Thus a LID attribute value may be updated even though a query does not explicitly affect a lid attribute in that tuple.

Updates are fully supported in ordered relations with the following side effects.

Appends - If the user specifies a lid value, the tuple is inserted in front of the tuple with that lid value. Thus all lid values following that tuple are incremented by one. If a lid is not specified and it corresponds to the lowest lid level (ie lid3 in a 3-dimensional ordered relation), the tuple will be inserted at the end of the lid subtree that it corresponds to. Otherwise the lid value is assumed to be "0" which indicates to the system that a new lid subtree will be created at level n where the lid value was specified.

Deletes - The user can delete tuples by specifying lid values. The side effect is that lid values will be collapsed due to the removed tuples.

Replaces - Like appends, new tuples will be inserted in front of the tuple with the old lid value that the user is trying to replace. If no new lid values are specified, they're assumed to be the old ones. To create a new lid subtree using replace, a lid value is "0" is to be specified.

SEE ALSO

modify(quel)

NAME

permit – add permissions to a relation

SYNOPSIS

define permit oplist { **on** | **of** | **to** } var [(attlist)]
to name [**at** term] [**from** time **to** time]
[**on** day **to** day] [**where** qual]

DESCRIPTION

The *permit* statement extends the current permissions on the relation specified by *var*. *Oplist* is a comma separated list of possible operations, which can be **retrieve**, **replace**, **delete**, **append**, or **all**; *all* is a special case meaning all permissions. *Name* is the login name of a user or the word **all**. *Term* is a terminal name of the form 'tty*x*' or the keyword **all**; omitting this phrase is equivalent to specifying *all*. *Time*s are of the form 'hh:mm' on a twenty-four hour clock which limit the times of the day during which this permission applies. *Day*s are three-character abbreviations for days of the week. The *qual* is appended to the qualification of the query when it is run.

Separate parts of a single *permit* statement are conjoined (ANDed). Different *permit* statements are disjoined (ORed). For example, if you include

. . . to eric at tty4 . . .

the *permit* applies only to eric when logged in on tty4, but if you include two *permit* statements

. . . to eric at all . . .
. . . to all at tty4 . . .

then when eric logs in on tty4 he will get the union of the permissions specified by the two statements. If eric logs in on ttyd he will get only the permissions specified in the first *permit* statement, and if bob logs in on tty4 he will get only the permissions specified in the second *permit* statment.

The *permit* statement may only be issued by the owner of the relation. Although a user other than the DBA may issue a *permit* statement, it is useless because noone else can access her relations anyway.

Permit statements do not apply to the owner of a relation or to views.

The statements

```
define permit all on x to all
define permit retrieve of x to all
```

with no further qualification are handled as special cases and are thus particularly efficient.

EXAMPLES

```
range of e is employee
define permit retrieve of e (name, sal) to marc
        at ttyd from 8:00 to 17:00
        on Mon to Fri
        where e.mgr = "marc"

range of p is parts
define permit retrieve of e to all
```

SEE ALSO

destroy(quel)

NAME

print – print relation(s)

SYNOPSIS

print relname {, relname}

DESCRIPTION

Print displays the contents of each relation specified on the terminal (standard output). The formats for various types of domains can be defined by the use of switches when *ingres* is invoked. Domain names are truncated to fit into the specified width.

To print a relation one must either be the owner of the relation, or the relation must have "retrieve to all" or "all to all" permissions.

See ingres(quel) for details.

EXAMPLE

```
/* Print the emp relation */
  print emp
  print emp, parts
```

SEE ALSO

permit(quel), retrieve(quel), ingres(unix), printr(unix) handle long lines of output correctly – no wrap around.

Print should have more formating features to make printouts more readable.

Print should have an option to print on the line printer.

NAME

quel – **QUE**ry Language for INGRES

DESCRIPTION

The following is a description of the general syntax of QUEL. Individual QUEL statements and commands are treated separately in the document; this section describes the syntactic classes from which the constituent parts of QUEL statements are drawn.

1. Comments

A comment is an arbitrary sequence of characters bounded on the left by "/*" and on the right by "*/":

/* This is a comment */

2. Names

Names in QUEL are sequences of no more than 12 alphanumeric characters, starting with an alphabetic. Underscore (_) is considered an alphabetic. All upper-case alphabetics appearing anywhere except in strings are automatically and silently mapped into their lower-case counterparts.

3. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

abs	**all**	**and**
any	**append**	**ascii**
at	**atan**	**avg**
avgu	**by**	**concat**
copy	**cos**	**count**
countu	**create**	**define**
delete	**delim**	**destroy**
exp	**float4**	**float8**
from	**gamma**	**help**
in	**index**	**int1**
int2	**int4**	**integrity**
into	**is**	**log**
max	**min**	**mod**
modify	**not**	**of**
on	**onto**	**or**
permit	**print**	**range**
replace	**retrieve**	**save**
sin	**sqrt**	**sum**
sumu	**to**	**unique**
until	**unuse**	**use**
view	**where**	

4. Constants

There are three types of constants, corresponding to the three data types available in QUEL for data storage.

4.1. String constants

Strings in QUEL are sequences of no more than 255 arbitrary ASCII characters bounded by double quotes (" "). Upper case alphabetics within strings are accepted literally. Also, in order to imbed quotes within strings, it is necessary to prefix them with '\' . The same convention applies to '\' itself.

Only printing characters are allowed within strings. Non-printing characters (i.e. control characters) are converted to blanks.

4.2. Integer constants

Integer constants in QUEL range from -2,147,483,647 to +2,147,483,647. Integer constants beyond that range will be converted to floating point. If the integer is greater than 32,767 or less than -32,767 then it will be left as a two byte integer. Otherwise it is converted to a four byte integer.

4.3. Floating point constants

Floating constants consist of an integer part, a decimal point, and a fraction part or scientific notation of the following format:

{<dig>} [.<dig>] [e|E [+|-] {<dig>}]

Where <dig> is a digit, [] represents zero or one, {} represents zero or more, and | represents alternation. An exponent with a missing mantissa has a mantissa of 1 inserted. There may be no extra characters embedded in the string. Floating constants are taken to be double-precision quantities with a range of approximately -10^{38} to 10^{38} and a precision of 17 decimal digits.

5. Attributes

An attribute is a construction of the form:

variable.domain

Variable identifies a particular relation and can be thought of as standing for the rows or tuples of that relation. A variable is associated with a relation by means of a *range* statement. *Domain* is the name of one of the columns of the relation over which the variable ranges. Together they make up an attribute, which represents values of the named domain.

If the attribute is a string type, it can be qualified with the substring notation. The substring notation is explained later.

6. Operators

6.1 Arithmetic operators

Arithmetic operators take numeric type expressions as operands. Unary operators group right to left; binary operators group left to right. The operators (in order of descending precedence) are:

+,-	(unary) plus, minus
**	exponentiation
*,/	multiplication, division
+,-	(binary) addition, subtraction

Parentheses may be used for arbitrary grouping. Arithmetic overflow and divide by zero are not checked on integer operations. Floating point operations are checked for overflow, underflow, and divide by zero only if the appropriate machine hardware exists and has been enabled.

6.2 Arithmetic string operators

The operator + is a string concatenator, like the ΨU function *concat*; however, its syntax is cleaner and it is not limited to two arguments, but like its arithmetic counterpart, can be used without restriction. Its counterpart, -, is the string equivalent of the difference operator on sets, with the special property that only the first instance of the right hand side is deleted from the string. The binding properties of these two operators are exactly equivalent to the arithmetic plus and minus, which means that they can be used in conjunction with parentheses to form complex expressions.

These two operators are most useful when used with the substring notation.

7. Expressions (a_expr)

An expression is one of the following:

constant
attribute
functional expression
aggregate or aggregate function
a combination of numeric expressions and arithmetic operators

For the purposes of this document, an arbitrary expression will be refered to by the name *a_expr.*

8. Formats

Every *a_expr* has a format denoted by a letter (**c, i,** or **f,** for character, integer, or floating data types respectively) and a number indicating the number of bytes of storage occupied. Formats currently supported are listed below. The ranges of numeric types are indicated in parentheses.

c1 – c255	character data of length 1–255 characters
i1	1-byte integer (−128 to +127)
i2	2-byte integer (−32768 to +32767)
i4	4-byte integer (−2,147,483,648 to +2,147,483,647)
f4	4-byte floating (-10^{38} to $+10^{38}$, 7 decimal digit precision)
f8	8-byte floating (-10^{38} to $+10^{38}$, 17 decimal digit precision)

One numeric format can be converted to or substituted for any other numeric format.

9. Type Conversion.

When operating on two numeric domains of different types, INGRES converts as necessary to make the types identical.

When operating on an integer and a floating point number, the integer is converted to a floating point number before the operation. When operating on two integers of different sizes, the smaller is converted to the size of the larger. When operating on two floating point number of different size, the larger is converted to the smaller.

The following table summarizes the possible combinations:

	i1	i2	i4	f4	f8
i1 –	i1	i2	i4	f4	f8
i2 –	i2	i2	i4	f4	f8
i4 –	i4	i4	i4	f4	f8
f4 –	f4	f4	f4	f4	f4
f8 –	f8	f8	f8	f4	f8

INGRES provides five type conversion operators specifically for overriding the default actions. The operators are:

int1(a_expr)	result type i1
int2(a_expr)	result type i2
int4(a_expr)	result type i4
float4(a_expr)	result type f4
float8(a_expr)	result type f8

The type conversion operators convert their argument a_expr to the requested type. *A_expr* can be anything including character. If a character value cannot be converted, an error occures and processing is halted. This can happen only if the syntax of the character value is incorrect.

Overflow is not checked on conversion.

10. Target_list

A target list is a parenthesized, comma separated list of one or more elements , each of which must be of one of the following forms:

a) *result_attname* **is** *a_expr*

Result_attname is the name of the attribute to be created (or an already existing attribute name in the case of update statements.) The equal sign ("=") may be used interchangeably with **is.** In the case where *a_expr* is anything other than a single attribute, this form must be used to assign a result name to the expression.

b) *attribute*

In the case of a *retrieve,* the resultant domain will acquire the same name as that of the attribute being retrieved. In the case of update statements (*append, replace*), the relation being updated must have a domain with exactly that name.

Inside the target list the keyword **all** can be used to represent all domains. For example:

```
range of e is employee
retrieve (e.all) where e.salary > 10000
```

will retrieve all domains of employee for those tuples which satisfy the qualification. **All** can be used in the target list of a *retrieve* or an **append.** The domains will be inserted in their "create" order, that is, the same order they were listed in the *create* statement.

11. Comparison operators

Comparison operators take arbitrary expressions as operands.

<	(less than)
<=	(less than or equal)
>	(greater than)
>=	(greater than or equal)
=	(equal to)
!=	(not equal to)

They are all of equal precedence. When comparisons are made on character attributes, all blanks are ignored.

12. Logical operators

Logical operators take clauses as operands and group left-to-right:

not	(logical not; negation)
and	(logical and; conjunction)
or	(logical or; disjunction)

Not has the highest precedence of the three. **And** and **or** have equal precedence. Parentheses may be used for arbitrary grouping.

13. Qualification (qual)

A *qualification* consists of any number of clauses connected by logical operators. A clause is a pair of expressions connected by a comparison operator:

```
a_expr comparison_operator a_expr
```

Parentheses may be used for arbitrary grouping. A qualification may thus be:

clause
not *qual*
qual **or** *qual*
qual **and** *qual*
(*qual*)

14. Functional expressions

A *functional expression* consists of a function name followed by a parenthesized (list of) operand(s). Functional expressions can be nested to any level. In the following list of functions supported (*n*) represents an arbitrary numeric type expression. The format of the result is indicated on the right.

abs(*n*) – same as *n* (absolute value)
ascii(*n*) – character string (converts numeric to character)
atan(*n*) – f8 (arctangent)
concat(*a,b*) – character (character concatenation. See 16.2)
cos(*n*) – f8 (cosine)
exp(*n*) – f8 (exponential of *n*)
gamma(*n*) – f8 (log gamma)
log(*n*) – f8 (natural logarithm)
mod(*n,b*) – same as *b* (*n* modulo *b*. *n* and *b* must be i1, i2, or i4)
sin(*n*) – f8 (sine)
sqrt(*n*) – f8 (square root)

15. Aggregate expressions

Aggregate expressions provide a way to aggregate a computed expression over a set of tuples.

15.1. Aggregation operators

The definitions of the aggregates are listed below.

count – (i4) count of occurrences
countu – (i4) count of unique occurrences
sum – summation
sumu – summation of unique values
avg – (f8) average (sum/count)
avgu – (f8) unique average (sumu/countu)
max – maximum
min – minimum
any – (i2) value is 1 if any tuples satisfy the qualification, else it is 0

15.2. Simple aggregate

aggregation_operator (*a_expr* [**where** *qual*])

A simple aggregate evaluates to a single scalar value. *A_expr* is aggregated over the set of tuples satisfying the qualification (or all tuples in the range of the expression if no qualification is present). Operators *sum* and *avg* require numeric type *a_expr; count, any, max* and *min* permit a character type attribute as well as numeric type *a_expr.*

Simple aggregates are completely local. That is, they are logically removed from the query, processed separately, and replaced by their scalar value.

15.3. "any" aggregate

It is sometimes useful to know if any tuples satisfy a particular qualification. One way of doing this is by using the aggregate *count* and checking whether the return is zero or non-zero. Using *any* instead of *count* is more efficient since processing is stopped, if possible, the first time a tuple satisfies a qualification.

Any returns 1 if the qualification is true and 0 otherwise.

15.4. Aggregate functions

aggregation_operator (*a_expr* **by** *by_domain*
{, *by_domain*} [**where** *qual*])

Aggregate functions are extensions of simple aggregates. The *by* operator groups (i.e. partitions) the set of qualifying tuples by *by_domain* values. For more than one *by_domain,* the values which are grouped by are the concatenation of individual *by_domain* values. *A_expr* is as in simple aggregates. The aggregate function evaluates to a set of aggregate results, one for each partition into which the set of qualifying tuples has been grouped. The aggregate value used during evaluation of the query is the value associated with the partition into which the tuple currently being processed would fall.

Unlike simple aggregates, aggregate functions are not completely local. The *by_list,* which differentiates aggregate functions from simple aggregates, is global to the query. Domains in the *by_list* are automatically linked to the other domains in the query which are in the same relation.

Example:

```
/* retrieve the average salary for the employees
working for each manager */
range of e is employee
retrieve (e.manager, avesal=avg(e.salary by e.manager))
```

15.5 Aggregates on Unique Values.

It is occasionally necessary to aggregate on unique values of an expression. The *avgu, sumu,* and *countu* aggregates all remove duplicate values before performing the aggregation. For example:

```
count(e.manager)
```

would tell you how many occurrences of *e.manager* exist. But

```
countu(e.manager)
```

would tell you how many unique values of *e.manager* exist.

16. Special character operators

There are four special features which are particular to character domains.

16.1 Pattern matching characters

There are eleven characters which take on special meaning when used in character constants (strings):

*	matches any string of zero or more characters.
?	matches any single character.
[..]	matches any of characters in the brackets.
##1	matches any string of zero or more characters.
##2	matches any string of zero or more characters.
##3	matches any string of zero or more characters.
##4	matches any string of zero or more characters.
##5	matches any string of zero or more characters.
##6	matches any string of zero or more characters.
##7	matches any string of zero or more characters.
##8	matches any string of zero or more characters.
##9	matches any string of zero or more characters.
##0	matches all instances of strings between two occurances of ##0.

These characters can be used in any combination to form a variety of tests. For example:

```
where e.name = "##1Kalash##2Joe##4" - matches any occurance of "Kalash", followed
                by "Joe".
where e.name = "##0Ingres##0" - matches all occurances of "Ingres" within a line.
where e.name = "*" - matches any name.
where e.name = "E*" - matches any name starting with "E".
where e.name = "*ein" - matches all names ending with "ein"
where e.name = "*[aeiou]*" - matches any name with at least one vowel.
where e.name = "Allman?" - matches any seven character name starting with "Allman".
where e.name = "[A-J]*" - matches any name starting with A,B,..,J.
```

The special meaning of the pattern matching characters can be disabled by preceding them with a '\'. Thus "*" refers to the character "*". When the special characters appear in the target list they must be escaped. For example:

```
title = "\*\*\* ingres \*\*\*"
```

is the correct way to assign the string "*** ingres ***" to the domain "title".

16.1.1 Numbered Wildcards

The numbered wildcards are unique in that they may also appear in a target list, as well as in a qualification. Each unique numbered wildcard used retains the same value in both the target list and the qualification list. Thus a query such as

```
replace t(text = "##1the##2")
    where t.text = "##1THE##2"
```

will replace an occurence of "THE" in t.text with "the".

The special global wildcard ##0 when used in the query

```
replace t(text = "##0the##0")
    where t.text = "##0THE##0"
```

will replace all occurrences of "THE" with "the".

16.2 Concatenation

There is a concatenation operator which can form one character string from two. Its syntax is "concat(field1, field2)". The size of the new character string is the sum of the sizes of the original two. Trailing blanks are trimmed from the first field, the second field is concatenated and the remainder is blank padded. The result is never trimmed to 0 length, however. Concat can be arbitrarily nested inside other concats. For example:

```
name = concat(concat(x.lastname, ","), x.firstname)
```

will concatenate x.lastname with a comma and then concatenate x.firstname to that.

16.3 Ascii (numeric to character translation)

The *ascii* function can be used to convert a numeric field to its character representation. This can be useful when it is desired to compare a numeric value with a character value. For example:

```
retrieve ( ... )
        where x.chardomain = ascii(x.numdomain)
```

Ascii can be applied to a character value. The result is simply the character value unchanged. The numeric conversion formats are determined by the printing formats (see ingres(unix)).

16.4 Substring notation

Any string attribute can be broken up into into a smaller substring using the following substring operators.

```
variable.domain(X,Y)
variable.domain(X,y%
variable.domain%X,Y)
variable.domain%X,Y%
```

Each of the above represents a certain substring of *domain*, denoted by the endpoints X and Y. Whether the endpoints are to be included or not is determined by the parentheses (exclusion) and percent signs (inclusion).

X and *Y* (optional) consist of a required part with optional qualifiers. The required part can be any of the following:

a string
w (a word)
c (a character)
A user defined delimiter (see delim(quel))

The optional qualifiers are a preceding digit, *i*, which specifies to look for the *ith* occurence, and a trailing *$*, which specifies to search backwards from the end of the string.

The rules for searching are very simple. Without the *$*, the value of *X* chosen is the *ith* occur-

rance (the default value of *i* is one) from the left end of the string. The search for *Y*, if it is requested, starts after the end of *X*. A dollar sign, however, always specifies that the search start from the end of the string regardless of the value of *X*. For illustrative purposes, assume a text field to contain the following:

I saw the dog, the cat, and the duck take a walk.
Then the following constructs would have the attached values:

```
r.text(3w,2"the"%      dog, the cat, and the
r.text%2"the"$,$%      the cat, and the duck take a walk.
```

When combined with the arithemtic string operators this facility can the quite powerful. For example, to remove the dog from the sentence requires only the simple following query:

```
replace r(text = r.text - r.text%"the","the"))
```

Or perhaps only the baby duck was taking a walk:

```
replace r(text = r.text%"I","the"% + "baby" + r.text%"duck",$})
```

SEE ALSO

append(quel), delete(quel), delim(quel), range(quel), replace(quel), retrieve(quel), ingres(unix)

BUGS

The maximum number of variables which can appear in one query is 10.

Numeric overflow, underflow, and divide by zero are not detected.

When converting between numeric types, overflow is not checked.

NAME

range – declare a variable to range over a relation

SYNOPSIS

range of variable **is** relname

DESCRIPTION

Range is used to declare variables which will be used in subsequent QUEL statements. The *variable* is associated with the relation specified by *relname.* When the *variable* is used in subsequent statements it will refer to a tuple in the named relation. A range declaration remains in effect for an entire INGRES session (until exit from INGRES), until the variable is redeclared by a subsequent range statement, or until the relation is removed with the destroy command.

EXAMPLE

```
/* Declare tuple variable e to range over relation emp */
  range of e is emp
```

SEE ALSO

quel(quel), destroy(quel)

BUGS

Only 10 variable declarations may be in effect at any time. After the 10th range statement, the least recently referenced variable is re-used for the next range statement.

NAME

replace – replace values of domains in a relation

SYNOPSIS

replace tuple_variable (target_list) [**where** qual]

DESCRIPTION

Replace changes the values of the domains specified in the *target_list* for all tuples which satisfy the qualification *qual.* The *tuple_variable* must have been declared to range over the relation which is to be modified. Note that a tuple variable is required and not the relation name. Only domains which are to be modified need appear in the *target_list.* These domains must be specified as result_attnames in the *target_list* either explicitly or by default (see quel(quel)).

Numeric domains may be replaced by values of any numeric type (with the exception noted below). Replacement values will be converted to the type of the result domain.

Only the owner of a relation, or a user with replace pemission on the relation can do *replace.*

If the tuple update would violate an integrity constraint (see integrity(quel)), it is not done.

EXAMPLE

```
/* Give all employees who work for Smith a 10% raise */
  range of e is emp
  replace e(sal = 1.1 * e.sal) where e.mgr = "Smith"
```

SEE ALSO

integrity(quel), permit(quel), quel(quel), range(quel)

DIAGNOSTICS

Use of a numeric type expression to replace a character type domain or vice versa will produce diagnostics.

BUGS

NAME

retrieve – retrieve tuples from a relation

SYNOPSIS

retrieve [[**into**] relname] (target_list) [**where** qual]
retrieve unique (target_list) [**where** qual]

DESCRIPTION

Retrieve will get all tuples which satisfy the qualification and either display them on the terminal (standard output) or store them in a new relation.

If a relname is specified, the result of the query will be stored in a new relation with the indicated name. A relation with this name owned by the user must not already exist. The current user will be the owner of the new relation. The relation will have domain names as specified in the *target_list* result_attnames. The new relation will be saved on the system for seven days unless explicitly saved by the user until a later date.

If the keyword **unique** is present, tuples will be sorted on the first domain, and duplicates will be removed, before being displayed.

The keyword **all** can be used when it is desired to retrieve all domains.

If no result *relname* is specified then the result of the query will be displayed on the terminal and will not be saved. Duplicate tuples are not removed when the result is displayed on the terminal.

The format in which domains are printed can be defined at the time ingres is invoked (see ingres(unix)).

If a result relation is specified then the default procedure is to modify the result relation to an cheapsort storage structure removing duplicate tuples in the process.

If the default cheapsort structure is not desired, the user can override this at the time INGRES is invoked by using the **-r** switch (see ingres(unix)).

Only the relation's owner and users with retrieve permission may *retrieve* from it.

EXAMPLE

```
/* Find all employees who make more than their manager */
  range of e is emp
  range of m is emp
  retrieve (e.name) where e.mgr = m.name
           and e.sal > m.sal
/* Retrieve all domains for those who make more
       than the average salary */
  retrieve into temp (e.all) where e.sal > avg(e.sal)
/* retrieve employees's names sorted */
  retrieve unique (e.name)
```

SEE ALSO

modify(quel), permit(quel), quel(quel), range(quel), save(quel), ingres(unix)

DIAGNOSTICS

BUGS

NAME

save – save a relation until a date.

SYNOPSIS

save relname **until** month day year

DESCRIPTION

Save is used to keep relations beyond the default 7 day life span.

Month can be an integer from 1 through 12, or the name of the month, either abbreviated or spelled out.

Only the owner of a relation can *save* that relation. There is an INGRES process which typically removes a relation immediately after its expiration date has passed.

The actual program which destroys relations is called purge. It is not automatically run. It is a local decision when expired relations are removed.

System relations have no expiration date.

EXAMPLE

```
/* Save the emp relation until the end of February 1987 */
  save emp until feb 28 1987
```

SEE ALSO

create(quel), retrieve(quel), purge(unix)

NAME

use - specify a group of delimitors to be used
unuse - specify a group of delimitors to no longer be used

SYNOPSIS

use groupname
unuse groupname

DESCRIPTION

The use statement specifies a group of delimitors which are to be checked when a delimitor name is used in a query. The unuse statement specifies that the group of delimitors should no longer be checked. When a delimitor is specified in a query, the pattern to be associated with the name will be searched for among the list of delimitor groups which have been used.

EXAMPLE

```
define delim paper (word, "[A-Z]{a-z}")
use paper
unuse paper
```

SEE ALSO

delim(quel), destroy(quel)

NAME

view – define a virtual relation

SYNOPSIS

define view name (target-list) [**where** qual]

DESCRIPTION

The syntax of the *view* statement is almost identical to the *retrieve into* statment; however, the data is not retrieved. Instead, the definition is stored. When the relation *name* is later used, the query is converted to operate on the relations specified in the *target-list.*

All forms of retrieval on the view are fully supported, but only a limited set of updates are supported because of anomolies which can appear. Almost no updates are supported on views which span more than one relation. No updates are supported that affect a domain in the qualification of the view or that affect a domain which does not translate into a simple attribute.

In general, updates are supported if and only if it can be guaranteed (without looking at the actual data) that the result of updating the view is identical to that of updating the corresponding real relation.

The person who defines a view must own all relations upon which the view is based.

EXAMPLE

```
range of e is employee
range of d is dept
define view empdpt (ename = e.name, e.sal, dname = d.name)
        where e.mgr = d.mgr
```

SEE ALSO

retrieve(quel), destroy(quel)

NAME

copydb – create batch files to copy out a data base and restore it.

SYNOPSIS

copydb [**-u***name*] database full-path-name-of-directory [relation ...]

DESCRIPTION

Copydb creates two INGRES command files in the directory: **Copy.out**, which contains Quel instuctions which will copy all relations owned by the user into files in the named directory, and **copy.in**, which contains instructions to copy the files into relations, create indexes and do modifies. The files will have the same names as the relations with the users INGRES id tacked on the end. (The directory MUST NOT be the same as the data base directory as the files have the same names as the relation files.) The **-u** flag may be used to run *copydb* with a different user id. (The fact that *copydb* creates the copy files does not imply that the user can necessarily access the specified relation). If relation names are specified only those relations will be included in the copy files.

Copydb is written in Equel and will access the database in the usual manner. It does not have to run as the INGRES user.

EXAMPLE

```
chdir /mnt/mydir
copydb db /mnt/mydir/backup
ingres db <backup/copy.out
tp rl backup
rm -r backup

tp xl
ingres db <backup/copy.in
```

DIAGNOSTICS

Copydb will give self-explanatory diagnostics. If "too many indexes" is reported it means that more than ten indexes have been specified on one relation. The constant can be increased and the program recompiled. Other limits are set to the system limits.

BUGS

Copydb assumes that indexes which are ISAM do not need to be remodified. *Copydb* cannot tell if the relation was modified with a fillfactor or minpages specification. The **copy.in** file may be edited to reflect this.

NAME

creatdb – create a data base

SYNOPSIS

creatdb [**-u***name*] [**-e**] [**-m**] [±**c**] [±**q**] dbname

DESCRIPTION

Creatdb creates a new INGRES database, or modifies the status of an existing database. The person who executes this command becomes the Database Administrator (DBA) for the database. The DBA has special powers not granted to ordinary users.

Dbname is the name of the database to be created. The name must be unique among all INGRES users.

The flags ±**c** and ±**q** specify options on the database. The form +*x* turns an option on, while -*x* turns an option off. The **-c** flag turns off the concurrency control scheme (default on). The **+q** flag turns on query modification (default on).

Concurrency control should not be turned off except on databases which are never accessed by more than one user. This applies even if users do not share data relations, since system relations are still shared. If the concurrency control scheme is not installed in UNIX, or if the special file /dev/lock does not exist or is not accessible for read-write by INGRES, concurrency control acts as though it is off (although it will suddenly come on when the lock driver is installed in UNIX).

Query modification must be turned on for the protection, integrity, and view subsystems to work, however, the system will run slightly slower in some cases if it is turned on. It is possible to turn query modification on if it is already off in an existing database, but it is not possible to turn it off if it is already on.

Databases with query modification turned off create new relations with all access permitted for all users, instead of no access except to the owner, the default for databases with query modification enabled.

Database options for an existing database may be modified by stating the **-e** flag. The database is adjusted to conform to the option flags. For example:

```
creatdb -e +q mydb
```

turns query modification on for database "mydb" (but leaves concurrency control alone). Only the database administrator (DBA) may use the **-e** flag.

When query modification is turned on, new relations will be created with no access, but previously created relations will still have all access to everyone. The *destroy* command may be used to remove this global permission, after which more selective permissions may be specified with the *permit* command.

The INGRES user may use the **-u** flag to specify a different DBA: the flag should be immediately followed by the login name of the user who should be the DBA.

The **-m** flag specifies that the UNIX directory in which the database is to reside already exists. This should only be needed if the directory if a mounted file system, as might occur for a very large database. The directory must exist (as **.../data/base/***dbname*), must be mode 777, and must be empty of all files.

The user who executes this command must have the U_CREATDB bit set in the status field of her entry in the users file.

The INGRES superuser can create a file in **.../data/base** containing a single line which is the full pathname of the location of the database. The file must be owned by INGRES and be mode 600. When the database is created, it will be created in the file named, rather than in the directory **.../data/base.** For example, if the file **.../data/base/ericdb** contained the line

/mnt/eric/database

then the database called "ericdb" would be physically stored in the directory /mnt/eric/database rather than in the directory .../data/base/ericdb.

EXAMPLE

creatdb demo
creatdb -ueric -q erics_db
creatdb -e +q -c -u:av erics_db

FILES

.../files/dbtmplt7
.../files/data/base/*
.../files/datadir/* (for compatibility with previous versions)

SEE ALSO

demodb(unix), destroydb(unix), users(files), chmod(I), destroydb(quel), permit(quel)

DIAGNOSTICS

No database name specified.
: You have not specified the name of the database to create (or modify) with the command.

You may not access this database
: Your entry in the users file says you are not authorized to access this database.

You are not a valid INGRES user
: You do not have a users file entry, and can not do anything with INGRES at all.

You are not allowed this command
: The U_CREATDB bit is not set in your users file entry.

You may not use the -u flag
: Only the INGRES superuser may become someone else.

<name> does not exist
: With **-e** or **-m**, the directory does not exist.

<name> already exists
: Without either **-e** or **-m**, the database (actually, the directory) already exists.

<name> is not empty
: With the **-m** flag, the directory you named must be empty.

You are not the DBA for this database
: With the **-e** flag, you must be the database administrator.

NAME

destroydb – destroy an existing database

SYNOPSIS

destroydb [-s] [-m] dbname

DESCRIPTION

Destroydb will remove all reference to an existing database. The directory of the database and all files in that directory will be removed.

To execute this command the current user must be the database administrator for the database in question, or must be the INGRES superuser and have the **-s** flag stated.

The **-m** flag causes *destroydb* not to remove the UNIX directory. This is useful when the directory is a separate mounted UNIX file system.

EXAMPLE

destroydb demo
destroydb -s erics_db

FILES

.../data/base/*
.../datadir/* (for compatibility with previous versions)

SEE ALSO

creatdb(unix)

DIAGNOSTICS

invalid dbname – the database name specified is not a valid name.
you may not reference this database – the database may exist, but you do not have permission to do anything with it.
you may not use the -s flag – you have tried to use the -s flag, but you are not the INGRES superuser.
you are not the dba – someone else created this database.
database does not exist – this database does not exist.

NAME

equel – Embedded QUEL interface to C

SYNOPSIS

equel [–d] [–f] [–r] file.q ...

DESCRIPTION

Equel provides the user with a method of interfacing the general purpose programming language "C" with INGRES. It consists of the EQUEL pre-compiler and the EQUEL runtime library.

Compilation

The precompiler is invoked with the statement:

equel [<flags>] file1.q [<flags>] file2.q ...

where file*n*.q are the source input file names, which must end with **.q**. The output is written to the file "file*n*.c". As many files as wished may be specified.
The flags that may be used are:

–d Generate code to print source listing file name and line number when a run-time error occurs. This can be useful for debugging, but takes up process space. Defaults to off.

–f Forces code to be on the same line in the output file as it is in the input file to ease interpreting C diagnostic messages. EQUEL will usually try to get all C code lines in the output file on the same lines as they were in the input file. Sometimes it must break up queries into several lines to avoid C-preprocessor line overflows, possibly moving some C code ahead some lines. With the –f flag specified this will never happen and, though the line buffer may overflow, C lines will be on the right line. This is useful for finding the line in the source file that C error diagnostics on the output file refer to.

–r Resets flags to default values. Used to supress other flags for some of the files in the argument list.

The output files may than be compiled using the C compiler:

cc file1.c file2.c ... **–lq**

The **–lq** requests the use of the EQUEL object library.

All EQUEL routines and globals begin with the characters "II", and so all globals variables and procedure names of the form II*xxx* are reserved for use by EQUEL and should be avoided by EQUEL users.

Basic Syntax

EQUEL commands are indicated by lines which begin with a double pound sign ("##"). Other lines are simply copied as is. All normal INGRES commands may be used in EQUEL and have the same effect as if invoked through the interactive terminal monitor. Only retrieve commands with no result relation specified have a different syntax and meaning.

The format of retrieve without a result relation is modified to:

retrieve (C-variable = a_fcn { , C-variable = a_fcn })

optionally followed (immediately) by:

```
##      [ where qual ]
## {
        /* C-code */
## }
```

This statement causes the "C-code" to be executed once for each tuple retrieved, with the "C-variable"s set appropriately. Numeric values of any type are converted as necessary. No conversion is done between numeric and character values. (The normal INGRES *ascii* function may be used for this purpose.)

Also, the following EQUEL commands are permitted.

ingres [ingres flags] data_base_name

This command starts INGRES running, and directs all dynamically following queries to the database *data_base_name.* It is a run-time error to execute this command twice without an intervening "## exit", as well as to issue queries while an "## ingres" statement is not in effect. Each flag should be enclosed in quotes to avoid confusion in the EQUEL parser:

```
## ingres "-f4f10.2" "-i212" demo
```

exit

Exit simply exits from INGRES. It is equivalent to the **\q** command to the teletype monitor.

Parametrized Quel Statements

Quel statements with target lists may be "parametrized". This is indicated by preceding the statement with the keyword "param". The target list of a parametrized statement has the form:

(*tl_var, argv*)

where *tl_var* is taken to be a string pointer at execution time (it may be a string constant) and interpreted as follows. For any parametrized EQUEL statement except a retrieve without a result relation (no "into rel") (i.e. append, copy, create, replace, retrieve into) the string *tl_var* is taken to be a regular target list except that wherever a '%' appears a valid INGRES type (f4, f8, i2, i4, c) is expected to follow. Each of these is replaced by the value of the corresponding entry into *argv* (starting at 0) which is interpreted to be a pointer to a variable of the type indicated by the '%' sequence. Neither *argv* nor the variables which it points to need be declared to EQUEL. For example:

```
char    *argv[10];

        argv[0] = &double_var;
        argv[1] = &int_var;
##      param append to rel
##              ("dom1 = %f8, dom2 = %i2", argv)
##      /* to escape the "%<ingres_type>" mechanism use "%%" */
##      /* This places a single '%' in the string. */
```

On a retrieve to C-variables, within *tl_var,* instead of the C-variable to retrieve into, the same '%' escape sequences are used to denote the type of the corresponding argv entry into which the value will be retrieved.

The qualification of any query may be replaced by a string valued variable, whose contents is interpreted at run time as the text of the qualification.

The *copy* statement may also be parametrized. The form of the parametrized *copy* is analogous to the other parametrized statements: the target list may be parametrized in the same manner as the *append* statements, and furthermore, the **from/into** keyword may be replaced by a string valued variable whose content at run time should be **into** or **from.**

Declarations

Any valid C variable declaration on a line beginning with a "##" declares a C-variable that may be used in an EQUEL statement and as a normal variable. All variables must be declared before being used. Anywhere a constant may appear in an INGRES command, a C-variable may appear. The value of the C-variable is substituted at execution time.

Neither nested structures nor variables of type *char* (as opposed to pointer to char or array of char) are allowed. Furthermore, there are two restrictions in the way variables are referenced within EQUEL statements. All variable usages must be dereferenced and/or subscripted (for arrays and pointers), or selected (for structure variables) to yield lvalues (scalar values). *Char* variables are used by EQUEL as a means to use *strings.* Therefore when using a *char* array or

pointer it must be dereferenced only to a "*char* *". Also, variables may not have parentheses in their references. For example:

```
## struct xxx
## {
        int     i;
##      int     *ip;
## }    **struct_var;

/* not allowed */
##      delete p where p.ifield = *(*struct_var)->ip

/* allowed */
##      delete p where p.ifield = *struct_var[0]->ip
```

C variables declared to EQUEL have either global or local scope. Their scope is local if their declaration is within a free (not bound to a retrieve) block declared to EQUEL. For example:

```
/* globals scope variable */
## int  Gint;

func(i)
int     i;
## {
        /* local scope variable */
##      int     *gintp;
        ...
## }
```

If a variable of one of the char types is used almost anywhere in an EQUEL statement the content of that variable is used at run time. For example:

```
##      char    *dbname[MAXDATABASES + 1];
        int     current_db;

        dbname[current_db] = "demo";
##      ingres dbname[current_db]
```

will cause INGRES to be invoked with data base "demo". However, if a variable's name is to be used as a constant, then the non-referencing operator '#' should be used. For example:

```
## char *demo;

        demo = "my_database";

        /* ingres -d my_database */
##      ingres "-d" demo

        /* ingres -d demo */
##      ingres "-d" #demo
```

The C-preprocessor's #include feature may be used on files containing equel statements and declarations if these files are named *anything*.**q.h.** An EQUEL processed version of the file, which will be #included by the C-preprocessor, is left in *anything*.**c.h.**

Errors and Interrupts

INGRES and run-time EQUEL errors cause the routine **IIerror** to be called, with the error number and the parameters to the error in an array of string pointers as in a C language main routine. The error message will be looked up and printed. before printing the error message, the routine (*IIprint_err)() is called with the error number that ocurred as its single argument. The error message corresponding to the error number returned by (*IIprint_err)() will be printed. Printing will be supressed if (*IIprint_err)() returns 0. IIprint_err may be reassigned to, and is useful for programs which map INGRES errors into their own error messages. In ad-

dition, if the "**-d**" flag was set the file name and line number of the error will be printed. The user may write an IIerror routine to do other tasks as long as the setting of IIerrflag is not modified as this is used to exit retrieves correctly.

Interrupts are caught by equel if they are not being ignored. This insures that the rest of INGRES is in sync with the EQUEL process. There is a function pointer, IIinterrupt, which points to a function to call after the interrupt is caught. The user may use this to service the interrupt. It is initialized to "exit()" and is called with -1 as its argument. For example:

```
extern int (*IIinterrupt)();
extern reset();

setexit();
IIinterrupt = reset;
mainloop();
```

To ignore interrupts, signal() should be called before the ## ingres satatement is executed.

FILES

.../files/error7_*
: Can be used by the user to decipher INGRES error numbers.

/lib/libq.a
: Run time library.

SEE ALSO

.../doc/other/equeltut.q, C reference manual, ingres(UNIX), quel(QUEL)

BUGS

The C-code embedded in the tuple-by-tuple retrieve operation may not contain additional QUEL statements or recursive invocations of INGRES.

There is no way to specify an **i1** format C-variable.

Includes of an equel file within a parameterized target list, or within a C variable's array subscription brackets, isn't done correctly.

NAME

geoquel – GEO-QUEL data display system

SYNOPSIS

geoquel [**-s**] [**-d**] [**-a**] [**-t***T*] [**-tn***T*] dbname

DESCRIPTION

GEO-QUEL is a geographical interface to INGRES.

Dbname is the name of an existing data base.

The format of the graphic output depends upon the type of terminal in use. GEO-QUEL will look up the terminal type at login time and produce output appropriate for that terminal. If the terminal in use is incapable of drawing graphic output then a display list is generated for a Tektronix 4014 but will only be displayed if the results are saved with *savemap* and then re-displayed.

The flags are interpreted as follows:

-s Don't print any of the monitor messages, including prompts. This is inclusive of the dayfile.

-d Don't print the dayfile.

-a Disable the autoclear function in the terminal monitor.

-t*T* Set the terminal type to *T*. *T* can be **gt40, gt42,** or **4014** for DEC's GT40-GT42, and Tektronix's 4014 respectively.

-tn*T* Do not display output but prepare the display list for a terminal of type *T*, where *T* is from the above list.

EXAMPLE

geoquel demo
geoquel -d demo
geoquel -s demo < batchfile

FILES

.../files/grafile7
.../files/ttytype

SEE ALSO

GEO-QUEL reference manual

DIAGNOSTICS

The diagnostics produced by GEO-QUEL are intended to be self-explanatory. Occasional messages may be produced by INGRES; for an explanation of these messages see the INGRES system documentation.

NAME

helpr – get information about a database.

SYNOPSIS

helpr [**-u***name*] [**±w**] database relation ...

DESCRIPTION

Helpr gives information about the named relation(s) out of the database specified, exactly like the *help* command.

Flags accepted are **-u** and **±u.** Their meanings are identical to the meanings of the same flags in INGRES.

SEE ALSO

ingres(unix), help(quel)

DIAGNOSTICS

bad flag – you have specified a flag which is not legal or is in bad format.

you may not access database – this database is prohibited to you based on status information in the users file.

cannot access database – the database does not exist.

NAME

ingres – INGRES relational data base management system

SYNOPSIS

ingres [*flags*] dbname [process_table]

DESCRIPTION

This is the UNIX command which is used to invoke INGRES. Dbname is the name of an existing data base. The optional flags have the following meanings (a "±" means the flag may be stated "+*x*" to set option *x* or "–*x*" to clear option *x*. "–" alone means that "–*x*" must be stated to get the *x* function):

±**U** Enable/disable direct update of the system relations and secondary indicies. You must have the 000004 bit in the status field of the users file set for this flag to be accepted. This option is provided for system debugging and is strongly discouraged for normal use.

–**u***name* Pretend you are the user with login name *name* (found in the users file). If *name* is of the form :*xx*, *xx* is the two character user code of a user. This may only be used by the DBA for the database or by the INGRES superuser.

–**c***N* Set the minimum field width for printing character domains to *N*. The default is 6.

–**i***lN* Set integer output field width to *N*. *l* may be 1, 2, or 4 for i1's, i2's, or i4's repectively.

–**f***lxM.N* Set floating point output field width to *M* characters with *N* decimal places. *l* may be 4 or 8 to apply to f4's or f8's respectively. *x* may be **e, E, f, F, g, G, n,** or **N** to specify an output format. **E** is exponential form, **F** is floating point form, and **G** and **N** are identical to **F** unless the number is too big to fit in that field, when it is output in **E** format. **G** format guarantees decimal point alignment; **N** does not. The default format for both is **n10.3.**

–**v***X* Set the column seperator for retrieves to the terminal and print commands to be *X*. The default is vertical bar.

–**r***M* Set modify mode on the *retrieve* command to *M*. *M* may be **isam, cisam, hash, chash, heap, cheap, heapsort,** or **cheapsort,** for ISAM, compressed ISAM, hash table, compressed hash table, heap, compressed heap, sorted heap, or compressed sorted heap. The default is "cheapsort".

–**n***M* Set modify mode on the *index* command to *M*. *M* can take the same values as the **–r** flag above. Default is "isam".

±**a** Set/clear the autoclear option in the terminal monitor. It defaults to set.

±**b** Set/reset batch update. Users must the 000002 bit set in the status field of the users file to clear this flag. This flag is normally set. When clear, queries will run slightly faster, but no recovery can take place. Queries which update a secondary index automatically set this flag for that query only.

±**d** Print/don't print the dayfile. Normally set.

±**s** Print/don't print any of the monitor messages, including prompts. This flags is normally set. If cleared, it also clears the **–d** flag.

±**w** Wait/don't wait for the database. If the **+w** flag is present, INGRES will wait if certain processes are running (purge,restore, and/or sysmod) on the given data base. Upon completion of those processes INGRES will proceed. If the **–w** flag is present, a message is returned and execution stopped if the data base is not available. If the **±w** flag is omitted and the data base is unavailable, the error message is returned if INGRES is running in foreground (more precisly if the standard input is from a terminal), otherwise the wait option is invoked.

Process_table is the pathname of a UNIX file which may be used to specify the run-time configuration of INGRES. This feature is intended for use in system maintenance only, and its unenlightened use by the user community is strongly discouraged.

Note: It is possible to run the monitor as a batch-processing interface using the '<', '>' and '|' operators of the UNIX shell, provided the input file is in proper monitor-format.

EXAMPLE

```
ingres demo
ingres -d demo
ingres -s demo < batchfile
ingres -f4g12.2 -i13 +b -rhash demo
```

FILES

.../files/users – valid INGRES users
.../data/base/* – data bases
.../datadir/* – for compatability with previous versions
.../files/proctab7 – runtime configuration file

SEE ALSO

monitor(quel)

DIAGNOSTICS

Too many options to INGRES – you have stated too many flags as INGRES options.

Bad flag format – you have stated a flag in a format which is not intelligible, or a bad flag entirely.

Too many parameters – you have given a database name, a process table name, and "something else" which INGRES doesn't know what to do with.

No database name specified

Improper database name – the database name is not legal.

You may not access database *name* – according to the users file, you do not have permission to enter this database.

You are not authorized to use the *flag* flag – the flag specified requires some special authorization, such as a bit in the users file, which you do not have.

Database *name* does not exist

You are not a valid INGRES user – you have not been entered into the users file, which means that you may not use INGRES at all.

You may not specify this process table – special authorization is needed to specify process tables.

Database temporarily unavailable – someone else is currently performing some operation on the database which makes it impossible for you to even log in. This condition should disappear shortly.

NAME

printr – print relations

SYNOPSIS

printr [*flags*] database relation ...

DESCRIPTION

Printr prints the named relation(s) out of the database specified, exactly like the *print* command. Retrieve permission must be granted to all people to execute this command.

Flags accepted are **–u, ±w, –c, –i, –f,** and **–v.** Their meanings are identical to the meanings of the same flags in INGRES.

SEE ALSO

ingres(unix), print(quel)

DIAGNOSTICS

bad flag – you have specified a flag which is not legal or is in bad format.
you may not access database – this database is prohibited to you based on status information in the users file.
cannot access database – the database does not exist.

NAME

purge – destroy all expired and temporary relations

SYNOPSIS

purge [**-f**] [**-p**] [**-a**] [**-s**] [**±w**] [database ...]

DESCRIPTION

Purge searches the named databases deleting system temporary relations. When using the **-p** flag, expired user relations are deleted. The **-f** flag will cause unrecognizable files to be deleted, normally purge will just report these files.

Only the database administrator (the DBA) for a database may run purge, except the INGRES superuser may purge any database by using the **-s** flag.

If no databases are specified all databases for which you are the DBA will be purged. All databases will be purged if the INGRES superuser has specified the **-s** flag. The **-a** flag will cause purge to print a message about the pending operation and execute it only if the response if a 'y'. Any other response is interpreted as "no".

Purge will lock the data base while it is being processed, since errors may occur if the database is active while purge is working on the database. If a data base is busy *purge* will report this and go on to the next data base, if any. If standard input is not a terminal *purge* will wait for the data base to be free. If **-w** flag is stated *purge* will not wait, regardless of standard input. The **+w** flag causes *purge* to always wait.

EXAMPLES

purge -p +w tempdata
purge -a -f

SEE ALSO

save(quel), restore(unix)

DIAGNOSTICS

who are you? – you are not entered into the users file.
not ingres superuser – you have tried to use the **-s** flag but you are not the INGRES superuser.
you are not the dba – you have tried to purge a database for which you are not the DBA.
cannot access database – the database does not exist.

BUGS

If no database names are given, only the databases located in the directory **data/base** are purged, and not the old databases in **datadir.** Explicit database names still work for databases in either directory.

NAME

restore – recover from an INGRES or UNIX crash.

SYNOPSIS

restore [-a] [-s] [±w] [database ...]

DESCRIPTION

Restore is used to restore a data base after an INGRES or UNIX crash. It should always be run after any abnormal termination to ensure the integrity of the data base.

In order to run restore, you must be the DBA for the database you are restoring or the INGRES superuser and specify the **-s** flag.

If no databases are specified then all databases for which you are the DBA are restored. All databases will be restored if the INGRES superuser has specified the **-s** flag.

If the **-a** flag is specified you will be asked before restore takes any serious actions. It is advisable to use this flag if you suspect the database is in bad shape. Using /dev/null as input with the **-a** flag will provide a report of problems in the data base. If there were no errors while restoring a database, *purge* will be called, with the same flags that were given to *restore,* to remove unwanted files and system temporaries. *Restore* may be called with the **-f** and/or **-p** flags for *purge.* Unrecognized files and expired relations are not removed unless the proper flags are given. In the case of an incomplete destroy, create or index *restore* will not delete files for partially created or destroyed relations. *Purge* must be called with the **-f** flag to accomplish this.

Restore locks the data base while it is being processed. If a data base is busy *restore* will report this and go on to the next data base. If standard input is not a terminal *restore* will wait for the data base to be free. If the **-w** flag is set *restore* will not wait regardless of standard input. If **+w** is set it will always wait.

Restore can recover a database from an update which had finished filling the batch file. Updates which did not make it to this stage should be rerun. Similarly modifies which have finished recreating the relation will be completed (the relation relation and attribute relations will be updated). If a destroy was in progress it will be carried to completion, while a create will almost always be backed out. Destroying a relation with an index should destroy the index so *restore* may report that a secondary relation has been found with no primary.

If interrupt (signal 2) is received the current database is closed and the next, if any, is processed. Quit (signal 3) will cause restore to terminate.

EXAMPLE

```
restore -f demo
restore -a grants < /dev/null
```

DIAGNOSTICS

All diagnostics are followed by a tuple from a system relations.

"No relation for attribute(s)" – the attributes listed have no corresponding entry in the relation relation

"No primary relation for index" – the tuple printed is the relation tuple for a secondary index for which there is no primary relation. The primary probably was destroyed the secondary will be.

"No indexes entry for primary relation" – the tuple is for a primary relation, the relindxd domain will be set to zero. This is the product of an incomplete destroy.

"No indexes entry for index" – the tuple is for a secondary index, the index will be destroyed. This is the product of an incomplete destroy.

"*relname* is index for" – an index has been found for a primary which is not marked as indexed. The primary will be so marked. This is probably the product of an incomplete index command. The index will have been created properly but not modified.

"No file for" – There is no data for this relation tuple, the tuple will be deleted. If, under the **-a** option, the tuple is not deleted purge will not be called.

"No secondary index for indexes entry" – An entry has been found in the indexes relation for which the secondary index does not exist (no relation relation tuple). The entry will be deleted.

SEE ALSO

purge(unix)

BUGS

If no database names are given, only the databases located in the directory **data/base** are restored, and not the old databases in **datadir.** Explicit database names still work for databases in either directory.

NAME

sysmod – modify system relations to predetermined storage structures.

SYNOPSIS

sysmod [-s] [-w] dbname **[relation] [attribute] [indexes] [tree] [protect] [integrities]**

DESCRIPTION

Sysmod will modify the relation, attribute, indexes, tree, protect, and integrities relations to hash unless at least one of the **relation, attribute, indexes, tree, protect,** or **integrities** parameters are given, in which case only those relations given as parameters are modified. The system relations are modified to gain maximum access performance when running INGRES. The user must be the data base administrator for the specified database, or be the INGRES superuser and have the **-s** flag stated.

Sysmod should be run on a data base when it is first created and periodically thereafter to maintain peak performance. If many relations and secondary indices are created and/or destroyed, *sysmod* should be run more often.

If the data base is being used while *sysmod* is running, errors will occur. Therefore, *sysmod* will lock the data base while it is being processed. If the data base is busy, *sysmod* will report this. If standard input is not a terminal *sysmod* will wait for the data base to be free. If **-w** flag is stated *sysmod* will not wait, regardless of standard input. The **+w** flag causes *sysmod* to always wait.

The system relations are modified to hash; the relation relation is keyed on the first domain, the indexes, attribute, protect, and integrities relations are keyed on the first two domains, and the tree relation is keyed on domains one, two, and five. The relation and attribute relations have the minpages option set at 10, the indexes, protect, and integrities relations have the minpages value set at 5.

SEE ALSO

modify(quel)

NAME

usersetup – setup users file

SYNOPSIS

.../bin/usersetup [flags [pathname]]

DESCRIPTION

The **/etc/passwd** file is read and reformatted to become the INGRES users file, stored into **.../files/users.** If *pathname* is specified, it replaces "...". If pathname is "–", the result is written to the standard output.

The user name, user, and group id's are initialized to be identical to the corresponding entry in the **/etc/passwd** file. The status field is initialized to be 000001, except for user **ingres,** which is initialized to all permission bits set. If the *status* parameter is provided, the field is set to this instead. The "initialization file" parameter is set to the file **.ingres** in the user's login directory. The user code field is initialized with sequential two-character codes. All other fields are initialized to be null.

After running *usersetup,* the **users** file must be edited. Any users who are to have any special authorizations should have the status field changed, according to the specifications in users(files). To disable a user from executing INGRES entirely, completely remove her line from the users file.

As UNIX users are added or deleted from the **/etc/passwd** file, the **users** file will need to be editted to reflect the changes. For deleted users, it is only necessary to delete the line for that user from the **users** file. To add a user, you must assign that user a code in the form "aa" and enter a line in the users file in the form:

name:cc:uid:gid:status:flags:proctab:initfile::databases

where *name* is the user name (taken from the first field of the **/etc/passwd** file entry for this user), *cc* is the user code assigned, which must be exactly two characters long and must not be the same as any other existing user codes, *uid* and *gid* are the user and group ids (taken from the third and fourth fields in the **/etc/passwd** entry), *status* is the status bits for this user, normally 000000, *flags* are the default flags for INGRES (on a per-user basis), *proctab* is the default process table for this user (which defaults to **=proctab7**), and *databases* is a list of the databases this user may enter. If null, she may use all databases. If the first character is a dash ("–"), the field is a comma separated list of databases which she may not enter. Otherwise, it is a list of databases which she may enter.

The *databases* field includes the names of databases which may be created.

Usersetup may be executed only once, to initially create the **users** file.

FILES

.../files/users
/etc/passwd

SEE ALSO

ingres(unix), passwd(V), users(files)

BUGS

It should be able to bring the **users** file up to date.

NAME

.../files/dayfile7 – INGRES login message

DESCRIPTION

The contents of the dayfile reflect user information of general system interest, and is more or less analogous to **/etc/motd** in UNIX. The file has no set format; it is simply copied at login time to the standard output device by the monitor if the **–s** or **–d** options have not been requested. Moreover the dayfile is not mandatory, and its absence will not generate errors of any sort; the same is true when the dayfile is present but not readable.

NAME

.../files/dbtmplt7 – database template

DESCRIPTION

This file contains the template for a database used by *creatdb*. It has a set of entries for each relation to be created in the database. The sets of entries are separated by a blank line. Two blank lines or an end of file terminate the file.

The first line of the file is the database status and the default relation status, separated by a colon. The rest of the file describes relations. The first line of each group gives the relation name followed by an optional relation status, separated by a colon. The rest of the lines in each group are the attribute name and the type, separated by a tab character.

All the status fields are given in octal, and have a syntax of a single number followed by a list of pairs of the form

±*x*±N

which says that if the ± *x* flag is asserted on the *creatdb* command line then set (clear) the bits specified by *N*.

The first set of entries must be for the relation catalog, and the second set must be for the attribute catalog.

EXAMPLE

```
3-c-1+q+2:010023
relation:-c-20
relid    c12
relowner      c2
relspec i1

attribute:-c-20
attrelid c12
attowner      c2
attname       c12

        (other relation descriptors)
```

SEE ALSO

creatdb(unix)

NAME

.../files/error7_? – files with INGRES errors

DESCRIPTION

These files contain the INGRES error messages. There is one file for each thousands digit; e.g., error number 2313 will be in file error7_2.

Each file consists of a sequence of error messages with associated error numbers. When an error enters the front end, the appropriate file is scanned for the correct error number. If found, the message is printed; otherwise, the first message parameter is printed.

Each message has the format

errnum <TAB> message tilde.

Messages are terminated by the tilde character ("~"). The message is scanned before printing. If the sequence **%*n*** is encountered (where *n* is a digit from 0 to 9), parameter *n* is substituted, where %0 is the first parameter.

The parameters can be in any order. For example, an error message can reference %2 before it references %0.

EXAMPLE

```
1003    line %0, bad database name %1~
1005    In the purge of %1, a bad %0 caused execution to halt~
1006    No process, try again.~
```

NAME

.../files/grafile7 – GEO-QUEL login message

DESCRIPTION

The contents of the grafile (GEO-QUEL dayfile) reflect user information of general system interest, and is more or less analogous to **/etc/motd** in UNIX. The file has no set format; it is simply copied at login time to the standard output device by GEO-QUEL if the **–s** or **–d** options have not been requested. Moreover the grafile is not mandatory, and its absence will not generate errors of any sort; the same is true when the grafile is present but not readable.

NAME

libq – Equel run-time support library

DESCRIPTION

Libq all the routines necessary for an equel program to load. It typically resides in **/usr/lib/libq.a**, and must be specified when loading equel pre-processed object code. It may be referenced on the command line of *cc* by the abbreviation **-lq**.

Several useful routines which are used by equel processes are included in the library. These may be employed by the equel programmer to avoid code duplication. They are:

```
int     IIatoi(buf, i)
char    *buf;
int     i;

char    *IIbmove(source, destination, len)
char    *source, *destination;
int     len;

char    *IIconcatv(buf, arg1, arg2, ..., 0)
char    *buf, *arg1, ...;

char    *IIitos(i)
int     i;

int     IIsèqual(s1, s2)
char    *s1, *s2;

int     IIlength(string)
char    *string;

IIsyserr(string, arg1, arg2, ...);
char    *string;
```

IIatoi	IIatoi is equivalent to atoi(UTIL).
IIbmove	Moves *len* bytes from *source* to *destination*, returning a pointer to the location after the last byte moved. Does not append a null byte.
IIconcatv	Concatenates into *buf* all of its arguments, returning a pointer to the null byte at the end of the concatenation. *Buf* may not be equal to any of the arg-n but arg1.
IIitos	IIitos is equivalent to itoa(III).
IIsequal	Returns 1 iff strings s1 is identical to s2.
IIlength	Returns max(length of *string* without null byte at end, 255)
IIsyserr	IIsyserr is diferrent from syserr(util) only in that it will print the name in IIproc_name, and in that there is no 0 mode. Also, it will always call exit(-1) after printing the error message.

There are also some global Equel variables which may be manipulated by the user:

```
int     IIerrflag;
char    *IImainpr;
char    (*IIprint_err)();
int     IIret_err();
int     IIno_err();
```

IIerrflag Set on an error from INGRES to be the error number (see the error message section of the "INGRES Reference Manual") that ocurred. This remains valid from the time the error occurrs to the time when the next equel statement is issued. This may be used just after an equel statement to see if it succeded.

IImainpr This is a string which determines which ingres to call when a "## ingres" is issued. Initially it is "/usr/bin/ingres".

IIprint_err This function pointer is used to call a function which determines what (if any) error message should be printed when an ingres error occurs. It is called from IIerror() with the error number as an argument, and the error message corresponding to the error number returned will be printed. If (*IIprint_err)(<*errno*>) returns 0, then no error message will be printed. Initially IIprint_err is set to IIret_err() to print the error that ocurred.

IIret_err Returns its single integer argument. Used to have (*IIprint_err)() cause printing of the error that ocurred.

IIno_err Returns 0. Used to have (*IIprint_err)() suppress error message printing. IIno_err is used when an error in a parametrized equel statemenr occurs to suppress printing of the corresponding parser error.

SEE ALSO

atoi(util), bmove(util), cc(I), equel(unix), exit(II), itoa(III), length(util), sequal(util), syserr(util)

NAME

.../files/proctab7 – INGRES runtime configuration information

DESCRIPTION

The process table describes the runtime configuration of the INGRES system. Each line of the process table has a special meaning depending on the first character of the line. Blank lines and lines beginning with an asterisk are comments. All other lines have a sequence of fields separated by commas. Pipe descriptor fields are lower case letters or digits; if they are digits they are replaced by file descriptors from the EQUEL flag or the @ flag.

D defines a macro. The first field is a single character macro name. The second field is the string to use as the value. Macros are expanded using "$*x*" where *x* is the macro name. The macro "P" is predefined to be the pathname of the INGRES subtree.

P introduces a process description. All lines up to an end of file or another P line describe a single process. The first field is the process number. The next field is the pathname of the binary to execute for this process. The third field is the name of the process to use for printing messages. The fourth field must be a single character lower case letter representing the input pipe that is normally read when nothing special is happening, or a vertical bar followed by a single digit, meaning to read from that file descriptor. The next field is a set of flags in octal regarding processing of this process; these are described below. The final field is a single letter telling what trace flag this process uses.

L defines what modules are defined locally by this process. The first field is the module number used internally. The second field is a set of flags describing processing of this module: the only bit defined is the 0001 bit which allows this module to be executed directly by the user. The third field is the function number in the process which defines this module. The final field is the module number to be executed after this module completes; zero is nothing (return).

R defines modules that are known to this process but which must be passed to another process for execution. The first field is the process number the modules will be found in. The second field is the pipe to write to get to that process. The third field is the pipe to read to get a response from that process. The fourth field is a set of flags: 0001 means to write the output pipe if you get a broadcast message, 0002 means that the process is physically adjacent on the read pipe, and 0004 means that the process is adjacent on the write pipe. The fifth and subsequent fields are the module numbers that are defined by this process.

The status bits for the **P** line are as follows:

000010 close diagnostic output
000004 close standard input
000002 run in user's directory, not database
000001 run as the user, not as INGRES

The lowest numbered process becomes the parent of all the other processes.

WARNING: Giving a user permission to specify his or her own process table will allow them to bypass all protection provided by INGRES. This facility should be provided for system debugging only!

EXAMPLE

The following example will execute a three process system.

```
DB:$P/bin
DS:$P/source
**** Process 0 – terminal monitor
P0:$B/monitor:MONITOR:h:0003:M
L0:0:0:0
R1:0:a:h:0007:1
* Process 1 – parser
P1:$B/parser:PARSER:a:0014:P
L3:1:0:0
```

```
R0:0:h:a:0006:0
R2:0:b:g:0007:5:6:7
* Process 2 -- data base utilities
P2:$B/alldbu:DBU:b:0014:Z
L5:0:6:0
L6:0:0:0
L7:0:1:0
R0:0:g:b:0000
R1:0:g:b:0006
```

NAME

.../files/startup – INGRES startup file

DESCRIPTION

This file is read by the monitor at login time. It is read before the user startup file specified in the users file. The primary purpose is to define a new editor and/or shell to call with the \e or \s commands.

SEE ALSO

monitor(quel), users(files)

NAME

.../files/ttytype – GEO-QUEL terminal type database

DESCRIPTION

The **ttytype** file describes each terminal on your system. GEO-QUEL will not attempt to display graphical output on terminals that are not capable of displaying it. There is a sample of the file in **.../geoquel/ttytype.sample.** This is a copy of the file in use on the Berkeley system.

The **ttytype** file consists of a series of lines; the first character is the terminal id, and the rest of the line tells the type of the terminal. The first of these characters is a terminal class, and the rest signify the brand, or some other more descriptive indication. A completely blank line terminates the useful part of the file, after which comments may appear unrestricted. In the sample file the currently recognized (defined) terminal types are listed as comments.

BUGS

The current version of GEO-QUEL only looks for terminals of graphic nature and therefore only the graphic terminals need be in this database. If any other system or sub-system wishes to use this file, all terminals must be kept up to date.

NAME

.../files/users – INGRES user codes and parameters

DESCRIPTION

This file contains the user information in fields seperated by colons. The fields are as follows:

* User name, taken directly from **/etc/passwd** file.
* User code, assigned by the INGRES super-user. It must be a unique two character code.
* UNIX user id. This MUST match the entry in the **/etc/passwd** file.
* UNIX group id. Same comment applies.
* Status word in octal. Bit values are:

0000001	creatdb permission
0000002	permits batch update override
0000004	permits update of system catalogs
0000020	can use trace flags
0000040	can turn off qrymod
0000100	can use arbitrary proctabs
0000200	can use the =proctab form
0100000	ingres superuser

* A list of flags automatically set for this user.
* The process table to use for this user.
* An initialization file to read be read by the monitor at login time.
* Unassigned.
* Comma seperated list of databases. If this list is null, the user may enter any database. If it begins with a '–', the user may enter any database except the named databases. Otherwise, the user may only enter the named databases.

Giving permission to a user to use arbitrary process tables is tantamount to turning off the protection system for that user.

EXAMPLE

```
ingres:aa:5:2:177777:–d:=special:/mnt/ingres/.ingres::
guest:ah:35:1:000000:::::demo,guest
```

SEE ALSO

initucode(util)

NAME

Error messages introduction

DESCRIPTION

This document describes the error returns which are possible from the INGRES data base system and gives an explanation of the probable reason for their occurrence. In all cases the errors are numbered *nxxx* where *n* indicates the source of the error, according to the following table:

1 = EQUEL preprocessor
2 = parser
3 = query modification
4 = decomposition and one variable query processor
5 = data base utilities
30 = GEO-QUEL errors

For a description of these routines the reader is referred to *The Design and Implementation of INGRES.* The *xxx* in an error number is an arbitrary identifier.

The error messages are stored in the file **.../files/error7_*n*,** where *n* is defined as above. The format of these files is the error number, a tab character, the message to be printed, and the tilde character ("~") to delimit the message.

In addition many error messages have "%*i*" in their body where *i* is a digit interpreted as an offset into a list of parameters returned by the source of the error. This indicates that a parameter will be inserted by the error handler into the error return. In most cases this parameter will be self explanatory in meaning.

Where the error message is thought to be completely self explanatory, no additional description is provided.

NAME

Parser error message summary

SYNOPSIS

Error numbers 2000 – 2999.

DESCRIPTION

The following errors can be generated by the parser. The parser reads your query and translates it into the appropriate internal form; thus, almost all of these errors indicate syntax or type conflict problems.

ERRORS

2000 %0 errors were found in quel program

2100 line %0, Attribute '%1' not in relation '%2'

This indicates that in a given line of the executed workspace the indicated attribute name is not a domain in the indicated relation.

2103 line %0, Function type does not match type of attribute '%1'

This error will be returned if a function expecting numeric data is given a character string or vice versa. For example, it is illegal to take the SIN of a character domain.

2106 line %0, Data base utility command buffer overflow

This error will result if a utility command is too long for the buffer space allocated to it in the parser. You must shorten the command or recompile the parser.

2107 line %0, You are not allowed to update this relation: %1

This error will be returned if you attempt to update any system relation or secondary index directly in QUEL (such as the RELATION relation). Such operations which compromise the integrity of the data base are not allowed.

2108 line %0, Invalid result relation for APPEND '%1'

This error message will occur if you execute an append command to a relation that does not exist, or that you cannot access. For example, append to junk(...) will fail if junk does not exist.

2109 line %0, Variable '%1' not declared in RANGE statement

Here, a symbol was used in a QUEL expression in a place where a tuple variable was expected and this symbol was not defined via a RANGE statement.

2111 line %0, Too many attributes in key for INDEX

A secondary index may have no more than 6 keys.

2117 line %0, Invalid relation name '%1' in RANGE statement

You are declaring a tuple variable which ranges over a relation which does not exist.

2118 line %0, Out of space in query tree - Query too long

You have the misfortune of creating a query which is too long for the parser to digest. The only options are to shorten the query or recompile the parser to have more buffer space for the query tree.

2119 line %0, MOD operator not defined for floating point or character attributes

The *mod* operator is only defined for integers.

2120 line %0, no pattern match operators allowed in the target list

Pattern match operators (such as "*") can only be used in a qualification.

2121 line %0, Only character type domains are allowed in CONCAT operator

2123 line %0, '%1.all' not defined for replace

2125 line %0, Cannot use aggregates ("avg" or "avgu") on character values
2126 line %0, Cannot use aggregates ("sum" or "sumu") on character values
2127 line %0, Cannot use numerical functions (ATAN, COS, GAMMA, LOG, SIN, SQRT, EXP, ABS) on character values
2128 line %0, Cannot use unary operators ("+" or "-") on character values
2129 line %0, Numeric operations (+ - * /) not allowed on character values

Many functions and operators are meaningless when applied to character values.

2130 line %0, Too many result domains in target list

Maximum number of result domains is MAXDOM (currently 49).

2132 line %0, Too many aggregates in this query

Maximum number of aggregates allowed in a query is MAXAGG (currently 49).

2133 line %0, Type conflict on relational operator

It is not legal to compare a character type to a numeric type.

2134 line %0, '%1' is not a constant operator.
Only 'dba' or 'usercode' are allowed.

2135 line %0, You cannot duplicate the name of an existing relation(%1)

You have tried to create a relation which would redefine an existing relation. Choose another name.

2136 line %0, There is no such hour as %1, use a 24 hour clock system
2137 line %0, There is no such minute as %1, use a 24 hour clock system
2138 line %0, There is no such time as 24:%1, use a 24 hour clock system

Errors 2136-38 indicate that you have used a bad time in a *permit* statement. Legal times are from 0:00 to 24:00 inclusive.

2139 line %0, Your database does not support query modification

You have tried to issue a query modification statement (*define*), but the database was created with the **-q** flag. To use the facilities made available by query modification, you must say:

creatdb -e +q *dbname*

to the shell.

2500 line %0, The word '%1', cannot follow this command

A 2500 error is reported by the parser if it cannot otherwise classify the error. One common way to obtain this error is to omit the required parentheses around the target list. The parser reports the last symbol which was obtained from the scanner. Sometimes, the last symbol is far ahead of the actual place where the error occurred. The string "EOF" is used for the last symbol when the parser has read past the query.

2501 line %0, The word '%1', cannot follow a RETRIEVE command
2502 line %0, The word '%1', cannot follow an APPEND command
2503 line %0, The word '%1', cannot follow a REPLACE command
2504 line %0, The word '%1', cannot follow a DELETE command
2507 line %0, The word '%1', cannot follow a DESTROY command
2508 line %0, The word '%1', cannot follow a HELP command
2510 line %0, The word '%1', cannot follow a MODIFY command
2511 line %0, The word '%1', cannot follow a PRINT command
2515 line %0, The word '%1', cannot follow a RETRIEVE UNIQUE command
2516 line %0, The word '%1', cannot follow a DEFINE VIEW command
2519 line %0, The word '%1', cannot follow a HELP VIEW, HELP INTEGRITY, or HELP PERMIT command
2522 line %0, The word '%1', cannot follow a DEFINE PERMIT command

2523 line %0, The word '%1', cannot follow a DEFINE INTEGRITY command
2526 line %0, The word '%1', cannot follow a DESTROY INTEGRITY or DESTROY PERMIT command

Errors 2502 through 2526 indicate that after an otherwise valid query, there was something which could not begin another command. The query was therefore aborted, since this could have been caused by misspelling **where** or something equally as dangerous.

2600 syntax error on line %0
last symbol read was: '%1'
2601 line %0, Syntax error on '%1', the correct syntax is:
RETRIEVE [[INTO]relname] (target_list) [WHERE qual]
RETRIEVE UNIQUE (target_list) [WHERE qual]
2602 line %0, Syntax error on '%1', the correct syntax is:
APPEND [TO] relname (target_list) [WHERE qual]
2603 line %0, Syntax error*a%1', the correct syntax is:
REPLACE tuple_variable (target_list) [WHERE qual]
2604 line %0, Syntax error on '%1', the correct syntax is:
DELETE tuple_variable [WHERE qual]
2605 line %0, Syntax error on '%1', the correct syntax is:
COPY relname (domname = format {, domname = format}) direction
2606 line %0, Syntax error on '%1', the correct syntax is:
CREATE relname (domname1 = format{, domname2 = format})
2607 line %0, Syntax error on '%1', the correct syntax is:
DESTROY relname {, relname}
DESTROY [PERMIT | INTEGRITY] relname [integer integer} | ALL]
2609 line %0, Syntax error on '%1', the correct syntax is:
INDEX ON relname IS indexname (domain1{, domain2})
2610 line %0, Syntax error on '%1', the correct syntax is:
MODIFY relname TO storage-structure [ON key1 [: sortord]
[{, key2 [:sortorder]}]] [WHERE [FILLFACTOR = n] [, MINPAGES = n] [, MAX-PAGES = n]]
2611 line %0, Syntax error on '%1', the correct syntax is:
PRINT relname{, relname}
2612 line %0, Syntax error on '%1', the correct syntax is:
RANGE OF variable IS relname
2613 line %0, Syntax error on '%1', the correct syntax is:
SAVE relname UNTIL month day year
2614 line %0, Syntax error on '%1', the correct syntax is:
DEFINE VIEW name (target list) [WHERE qual]
DEFINE PERMIT oplist {ON|OF|TO} var [(attlist)] TO name [AT term] [FROM time TO time] [ON day TO day] [WHERE qual]
DEFINE INTEGRITY ON var IS qual
2615 line %0, Syntax error on '%1', the correct syntax is:
RETRIEVE UNIQUE (target_list) [WHERE qual]
2616 line %0, Syntax error on '%1', the correct syntax is:
DEFINE VIEW name (target_list) [WHERE qual]
2619 line %0, Syntax error on '%1', the correct syntax is:
HELP VIEW relname[, relname]
HELP PERMIT relname[, relname]
HELP INTEGRITY relname[, relname]
2622 line %0, Syntax error on '%1', the correct syntax is:
DEFINE PERMIT oplist {ON|OF|TO} var [(attlist)] TO name [AT term] [FROM time TO time] [ON day TO day] [WHERE qual]
2623 line %0, Syntax error on '%1', the correct syntax is:
DEFINE INTEGRITY ON var IS qual

Errors 2600 through 2623 are generated when a command's syntax has been violated. The correct syntax is given. If the command cannot be determined, error 2600 is given.

2700 line %0, non-terminated string

You have omitted the required string terminator (").

2701 line %0, string too long

Somehow, you have had the persistence or misfortune to enter a character string constant longer than 255 characters.

2702 line %0, invalid operator

You have entered a character which is not alphanumeric, but which is not a defined operator, for example, "?".

2703 line %0, Name too long '%1'

In INGRES relation names and domain names are limited to 12 characters.

2704 line %0, Out of space in symbol table - Query too long

Your query is too big to process. Try breaking it up with more \go commands.

2705 line %0, non-terminated comment

You have left off the comment terminator symbol ("*/").

2707 line %0, bad floating constant: %1

Either your floating constant was incorrectly specified or it was too large or too small. Currently, overflow and underflow are not checked.

2708 line %0, control character passed in pre-converted string

In EQUEL a control character became embedded in a string and was not caught until the scanner was processing it.

2709 line %0, buffer overflow in converting a number

Numbers cannot exceed 256 characters in length. This shouldn't become a problem until number formats in INGRES are increased greatly.

2800 line %0, yacc stack overflow in parsing query

NAME

Query Modification error message summary

SYNOPSIS

Error numbers 3000 – 3999.

DESCRIPTION

These error messages are generated by the Query Modification module. These errors include syntactic and semantic problems from view, integrity, and protection definition, as well as run time errors – such as inability to update a view, or a protection violation.

ERRORS

3310 %0 on view %1: cannot update some domain

You tried to perform operation %0 on a view; however, that update is not defined.

3320 %0 on view %1: domain occurs in qualification of view

It is not possible to update a domain in the qualification of a view, since this could cause the tuple to disappear from the view.

3330 %o on view %1: update would result in more than one query

You tried to perform some update on a view which would update two underlying relations.

3340 %0 on view %1: views do not have TID's

You tried to use the Tuple IDentifier field of a view, which is undefined.

3350 %0 on view %1: cannot update an aggregate value

You cannot update a value which is defined in the view definition as an aggregate.

3360 %0 on view %1: that update might be non-functional

There is a chance that the resulting update would be non-functional, that is, that it may have some unexpected side effects. INGRES takes the attitude that it is better to not try the update.

3490 INTEGRITY on %1: cannot handle aggregates yet

You cannot define integrity constraints which include aggregates.

3491 INTEGRITY on %1: cannot handle multivariable constraints

You cannot define integrity constraints on more than a single variable.

3492 INTEGRITY on %1: constraint does not initially hold

When you defined the constraint, there were already tuples in the relation which did not satisfy the constraint. You must fix the relation so that the constraint holds before you can declare the constraint.

3493 INTEGRITY on %1: is a view

You can not define integrity constraints on views.

3494 INTEGRITY on %1: You must own '%1'

You must own the relation when you declare integrity constraints.

3500 %0 on relation %1: protection violation

You have tried to perform an operation which is not permitted to you.

3590 PERMIT: bad terminal identifier "%2"

In a *permit* statement, the terminal identifier field was improper.

3591 PERMIT: bad user name "%2"

You have used a user name which is not defined on the system.

3592 PERMIT: Relation '%1' not owned by you

You must own the relation before issuing protection constraints.

3593 PERMIT: Relation '%1' must be a real relation (not a view)

You can not define permissions on views.

3594 PERMIT on %1: bad day-of-week '%2'

The day-of-week code was unrecognized.

3595 PERMIT on %1: only the DBA can use the PERMIT statement

Since only the DBA can have shared relations, only the DBA can issue *permit* statements.

3700 Tree buffer overflow in query modification
3701 Tree build stack overflow in query modification

Bad news. An internal buffer has overflowed. Some expression is too large. Try making your expressions smaller.

NAME

One Variable Query Processor error message summary

SYNOPSIS

Error numbers 4000 – 4499.

DESCRIPTION

These error messages can be generated at run time. The One Variable Query Processor actually references the data, processing the tree produced by the parser. Thus, these error messages are associated with type conflicts detected at run time.

ERRORS

4100 ovqp query list overflowed

This error is produced in the unlikely event that the internal form of your interaction requires more space in the one variable query processor than has been allocated for a query buffer. There is not much you can do except shorten your interaction or recompile OVQP with a larger query buffer.

4106 the interpreters stack overflowed – query too long
4107 the buffer for ASCII and CONCAT commands overflowed

More buffer overflows.

4108 cannot use arithmetic operators on two character fields
4109 cannot use numeric values with CONCAT operator

You have tried to perform a numeric operation on character fields.

4110 floating point exception occurred.

If you have floating point hardware instead of the floating point software interpreter, you will get this error upon a floating point exception (underflow or overflow). Since the software interpreter ignores such exceptions, this error is only possible with floating point hardware.

4111 character value cannot be converted to numeric due to incorrect syntax.

When using int1, int2, int4, float4, or float8 to convert a character to value to a numeric value, the character value must have the proper syntax. This error will occur if the character value contained non-numeric characters.

4112 ovqp query vector overflowed

Similar to error 4100.

4113 compiler text space ran out
4114 compiler ran out of registers

These errors refer to an experimental version of the system that is not currently released.

4199 you must convert your 6.0 secondary index before running this query!

The internal format of secondary indices was changed between versions 6.0 and 6.1 of INGRES. Before deciding to use a secondary index OVQP checks that it is not a 6.0 index. The solution is to destroy the secondary index and recreate it.

NAME

Decomposition error message summary

SYNOPSIS

Error numbers 4500 – 4999.

DESCRIPTION

These error messages are associated with the process of decomposing a multi-variable query into a sequence of one variable queries which can be executed by OVQP.

ERRORS

4602 query involves too many relations to create aggregate function intermediate result.

In the processing of aggregate functions it is usually necessary to create an intermediate relation for *each* aggregate function. However, no query may have more than ten variables. Since aggregate functions implicitly increase the number of variables in the query, you can exceed this limit. You must either break the interaction apart and process the aggregate functions separately or you must recompile INGRES to support more variables per query.

4610 Query too long for available buffer space (qbufsize).
4611 Query too long for available buffer space (varbufsiz)
4612 Query too long for available buffer space (sqsiz)
4613 Query too long for available buffer space (stacksiz)
4614 Query too long for available buffer space (agbufsiz).

These will happen if the internal form of the interaction processed by decomp is too long for the available buffer space. You must either shorten your interaction or recompile decomp. The name in parenthesis gives the internal name of which buffer was too small.

4615 Aggregate function is too wide or has too many domains.

The internal form of an aggregate function must not contain more than 49 domains or be more than 1010 bytes wide. Try breaking the aggregate function into two or more parts.

4620 Target list for "retrieve unique" has more than 49 domains or is wider than 1010 bytes.

NAME

Data Base Utility error message summary

SYNOPSIS

Error numbers 5000 - 5999

DESCRIPTION

The Data Base Utility functions perform almost all tasks which are not directly associated with processing queries. The error messages which they can generate result from some syntax checking and a considerable amount of semantic checking.

ERRORS

5001 PRINT: bad relation name %0

You are trying to print a relation which doesn't exist.

5002 PRINT: %0 is a view and can't be printed

The only way to print a view is by retrieving it.

5003 PRINT: Relation %0 is protected.

You are not authorized to access this relation.

5102 CREATE: duplicate relation name %0

You are trying to create a relation which already exists.

5103 CREATE: %0 is a system relation

You cannot create a relation with the same name as a system relation. The system depends on the fact that the system relations are unique.

5104 CREATE %0: invalid attribute name %1

This will happen if you try to create a relation with an attribute longer than 12 characters.

5105 CREATE %0: duplicate attribute name %1

Attribute names in a relation must be unique. You are trying to create one with a duplicated name.

5106 CREATE %0: invalid attribute format "%2" on attribute %1

The allowed formats for a domain are c1–c255, i1, i2, i4, f4 and f8. Any other format will generate this error.

5107 CREATE %0: excessive domain count on attribute %1

A relation cannot have more than 49 domains. The origin of this magic number is obscure. This is very difficult to change.

5108 CREATE %0: excessive relation width on attribute %1

The maximum number of bytes allowed in a tuple is 1010. This results from the decision that a tuple must fit on one UNIX "page". Assorted pointers require the 14 bytes which separates 1010 from 1024. This "magic number" is very hard to change.

5201 DESTROY: %0 is a system relation

The system would immediately stop working if you were allowed to do this.

5202 DESTROY: %0 does not exist or is not owned by you

To destroy a relation, it must exist, and you must own it.

5203 DESTROY: %0 is an invalid integrity constraint identifier

Integers given do not identify integrity constraints on the specified relation. For example: If you were to type "destroy permit parts 1, 2, 3", and 1, 2, or 3 were not the numbers "help permit parts" prints out for permissions on parts, you would get this

error.

5204 DESTROY: %0 is an invalid protection constraint identifier

Integers given do not identify protection constraints on the specified relation. Example as for error 5203.

5300 INDEX: cannot find primary relation

The relation does not exist – check your spelling.

5301 INDEX: more than maximum number of domains

A secondary index can be created on at most six domains.

5302 INDEX: invalid domain %0

You have tried to create an index on a domain which does not exist.

5303 INDEX: relation %0 not owned by you

You must own relations to put indicies on them.

5304 INDEX: relation %0 is already an index

INGRES does not permit tertiary indicies.

5305 INDEX: relation %0 is a system relation

Secondary indices cannot be created on system relations.

5306 INDEX: %0 is a view and an index can't be built on it

Since views are not physically stored in the database, you cannot build indicies on them.

5401 HELP: relation %0 does not exist

5402 HELP: cannot find manual section "%0"

Either the desired manual section does not exist, or your system does not have any on-line documentation.

5403 HELP: relation %0 is not a view

Did a "help view" (which prints view definition) on a nonview. For example: "help view overpaidv" prints out overpaidv's view definition.

5404 HELP: relation %0 has no permissions on it granted
5405 HELP: relation %0 has no integrity constraints on it

You have tried to print the permissions or integrity constraints on a relation which has none specified.

5410 HELP: tree buffer overflowed
5411 HELP: tree stack overflowed

Still more buffer overflows.

5500 MODIFY: relation %0 does not exist

5501 MODIFY: you do not own relation %0

You cannot modify the storage structure of a relation you do not own.

5502 MODIFY %0: you may not provide keys on a heap

By definition, heaps do not have keys.

5503 MODIFY %0: too many keys provided

You can only have 49 keys on any relation.

5504 MODIFY %0: cannot modify system relation

System relations can only be modified by using the *sysmod* command to the shell; for example

sysmod *dbname*

5507 MODIFY %0: duplicate key "%1"

You may only specify a domain as a key once.

5508 MODIFY %0: key width (%1) too large for isam

When modifying a relation to isam, the sum of the width of the key fields cannot exceed 245 bytes.

5510 MODIFY %0: bad storage structure "%1"

The valid storage structure names are heap, cheap, isam, cisam, hash, and chash.

5511 MODIFY %0: bad attribute name "%1"

You have specified an attribute that does not exist in the relation.

5512 MODIFY %0: "%1" not allowed or specified more than once

You have specified a parameter which conflicts with another parameter, is inconsistant with the storage mode, or which has already been specified.

5513 MODIFY %0: fillfactor value %1 out of bounds

Fillfactor must be between 1 and 100 percent.

5514 MODIFY %0: minpages value %1 out of bounds

Minpages must be greater than zero.

5515 MODIFY %0: "%1" should be "fillfactor", "maxpages", or "minpages"

You have specified an unknown parameter to *modify*.

5516 MODIFY %0: maxpages value %1 out of bounds

5517 MODIFY %0: minpages value exceeds maxpages value

5518 MODIFY %0: invalid sequence specifier "%1" for domain %2.

Sequence specifier may be "ascending" (or "a") or "descending" (or "d") in a *modify*. For example:

```
modify parts to heapsort on
    pnum:ascending,
    pname:descending
```

5519 MODIFY: %0 is a view and can't be modified

Only physical relations can be modified.

5520 MODIFY: %0: sequence specifier "%1" on domain %2 is not allowed with the specified storage structure.

Sortorder may be supplied only when modifying to **heapsort** or **cheapsort.**

5600 SAVE: cannot save system relation "%0"

System relations have no save date and are guaranteed to stay for the lifetime of the data base.

5601 SAVE: bad month "%0"
5602 SAVE: bad day "%0"
5603 SAVE: bad year "%0"

This was a bad month, bad day, or maybe even a bad year for INGRES.

5604 SAVE: relation %0 does not exist or is not owned by you

5800 COPY: relation %0 doesn't exist

5801 COPY: attribute %0 in relation %1 doesn't exist or it has been listed twice

5803 COPY: too many attributes

Each dummy domain and real domain listed in the copy statement count as one attribute. The limit is 150 attributes.

5804 COPY: bad length for attribute %0. Length="%1"

5805 COPY: can't open file %0

On a copy "from", the file is not readable by the user.

5806 COPY: can't create file %0

On a copy "into", the file is not creatable by the user. This is usually caused by the user not having write permission in the specified directory.

5807 COPY: unrecognizable dummy domain "%0"

On a copy "into", a dummy domain name is used to insert certain characters into the unix file. The domain name given is not valid.

5808 COPY: domain %0 size too small for conversion.
There were %2 tuples successfully copied from %3 into %4

When doing any copy except character to character, copy checks that the field is large enough to hold the value being copied.

5809 COPY: bad input string for domain %0. Input was "%1". There were %2 tuples successfully copied from %3 into %4

This occurs when converting character strings to integers or floating point numbers. The character string contains something other than numeric characters (0-9,+,−,blank,etc.).

5810 COPY: unexpected end of file while filling domain %0.
There were %1 tuples successfully copied from %2 into %3

5811 COPY: bad type for attribute %0. Type="%1"

The only accepted types are i, f, c, and d.

5812 COPY: The relation "%0" has a secondary index. The index(es) must be destroyed before doing a copy "from"

Copy cannot update secondary indices. Therefore, a copy "from" cannot be done on an indexed relation.

5813 COPY: You are not allowed to update the relation %0

You cannot copy into a system relation or secondary index.

5814 COPY: You do not own the relation %0.

You cannot use copy to update a relation which you do not own. A copy "into" is allowed but a copy "from" is not.

5815 COPY: An unterminated "c0" field occurred while filling domain %0. There were %1 tuples successfully copied from %2 into %3

A string read on a copy "from" using the "c0" option cannot be longer than 1024 characters.

5816 COPY: The full pathname must be specified for the file %0

The file name for copy must start with a "/".

5817 COPY: The maximum width of the output file cannot exceed 1024 bytes per tuple

The amount of data to be output to the file for each tuple exceeds 1024. This usually happens only if a format was mistyped or a lot of large dummy domains were specified.

5818 COPY: %0 is a view and can't be copied

Only physical relations can be copied.

5819 COPY: Warning: %0 duplicate tuples were ignored.

On a copy "from", duplicate tuples were present in the relation.

5820 COPY: Warning: %0 domains had control characters which were converted to blanks.

5821 COPY: Warning: %0 c0 character domains were truncated.

Character domains in c0 format are of the same length as the domain length. You had a domain value greater than this length, and it was truncated.

5822 COPY: Relation %0 is protected.

You are not authorized to access this relation.